POLITECNICO DI TORINO
Repository ISTITUZIONALE

FireNN: Neural Networks Reliability Evaluation on Hybrid Platforms

Original
FireNN: Neural Networks Reliability Evaluation on Hybrid Platforms / De Sio, C., Azimi, S., Sterpone, L.. - In: IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTING. - ISSN 2168-6750. - ELETTRONICO. - 10:2(2022), pp. 549-563. [10.1109/TETC.2022.3152668]

Availability:
This version is available at: 11583/2958120 since: 2022-03-11T12:09:06Z

Publisher:
IEEE Computer Society

Published
DOI:10.1109/TETC.2022.3152668

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

©2022 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)

14 October 2026

FireNN: Neural Networks Reliability Evaluation on Hybrid Platforms

C. De Sio, *Student Member, IEEE*, S. Azimi, *Member, IEEE,* L. Sterpone, *Member, IEEE*

Abstract— Modern neural network complexity has grown dramatically in recent years, leading to the adoption of hardware-accelerated solutions to cope with the computational power required by the new network architectures. The possibility to adapt the network size and performance to different platforms enhanced the interests of safety-critical applications such as automotive and avionic. Today, the reliability evaluation of neural networks is still premature and requires platforms to measure the safety standards required by mission-critical applications. For this reason, the interest in studying the reliability of neural networks is growing. In this work, we propose a new approach for evaluating the resiliency of neural networks by using programmable hardware of hybrid platforms. The approach relies on the reconfigurable hardware for emulating the target hardware platform and performing the fault injection process. The main advantage of the proposed approach is to involve the on-hardware execution of the neural network in the reliability analysis without modifying the hardware implementation of the network under analysis, and addressing specific fault models. The implementation of FireNN, the platform based on the proposed approach is detailly described in the paper. Experimental analyses are performed using fault injection on the AlexNet Convolutional Neural Network. The analyses are carried out by means of the FireNN platform and the obtained results are compared with the outcome of traditional software-level evaluations. Results are commented taking into account the insight into the hardware level achieved by using the FireNN platform.

Index Terms— Deep Neural Network, Fault injection, FPGA, Hardware Emulation, Reliability

1 Introduction

NEURAL networks have dramatically risen in importance during recent years. Their outstanding performance in solving complex prediction and classification problems made them a ubiquitous technology, adopted in many fields, such as healthcare, automotive, speech recognition, natural language processing, aerospace, and many others [1]–[4]. The diffusion of this technology in such a wide range of applications led to the steady growth of research interest around neural networks. The excitement on this technology both on the research and industry sides is resulting in a succession of new architectures and solutions aiming to fulfill different requirements. In particular, the improvement of neural network accuracy and performance has been the main topic on which the researchers focused. Convolutional Neural Networks (CNNs) have proven to be one of the most promising families of neural networks for visual tasks, with astonishing results in terms of accuracy and performance [5]. However, new challenges have come along with the progress introduced by CNNs. The huge amount of computational power and memory demanded by modern neural network architectures made the use of traditional platforms, as CPUs, used for the training and inferring phases unfeasible. Consequently, the use of hardware accelerators able to provide a huge computational power, in particular exploiting the parallelism of a large number of computational logic units, has become the conventional solution. The set of hardware used as neural network accelerators is composed of different solutions as well. Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs), and Application-Specific Integrated Circuits (ASICs) are widely used as hardware-accelerated platforms for running neural networks [6], [7]. The impressive results in terms of performance showed by hardware-accelerated neural networks are very appealing for mission-critical applications too [8]–[10]. In particular, the capabilities of decision making, intelligent control, and visual processing are of invaluable value for autonomous driving systems in the automotive and avionic sectors, as well as deep space exploration. Due to the criticality intrinsic to mission-critical applications, where a failure can jeopardize human lives or huge investments, the reliability of these systems is rising in importance as a metric in their design. Even if the improvement of performance is the leading path in neural network research, the new safety standards requirements, such as ISO-26262 and DO-254, are pushing efforts towards the study on the resilience of these systems [11], [12]. Moreover, the continuous evolution of state-of-the-art neural network topologies and hardware platforms, along with the rising complexity of modern neural network architectures, makes the evaluation and analysis of the reliability and resilience of these systems against faults problematic.

Neural network architecture and the hardware platforms supporting their executions are heterogeneous. Therefore, to address the reliability evaluation of neural networks, it is necessary to adopt a flexible platform capable to address different types of fault models while evaluating the overall network behavior during the inference execution. In detail, the hardware is subjected to an ample set of phenomena, accentuated by the increase in the size

C. De Sio, S. Azimi and L. Sterpone are with the Dipartimento di Automatica e Informatica, Politecnico di Torino, Torino, Italy, 10138.
E-mail:
corrado.desio@polito.it, sarah.azimi@polito.it, luca.sterpone@polito.it

xxxx-xxxx/0x/$xx.00 © 200x IEEE Published by the IEEE Computer Society

downscaling, that can produce faults that propagate to the application level, provoking errors, or failures in the whole system. The source of these faults comes both from the device itself, such as faults deriving from aging or manufacturing process variations, and external factors, such as transient and permanent errors generated by ionizing radiation. Moreover, neural network reliability and resilience are strongly influenced by the fault location, the design choices, such as data precisions, activation functions, and algorithms. Hence, they depend on network architecture and topology but the implementation of the network on the specific hardware accelerator and the hardware itself also play central roles [13], [14]. With so many variables to consider, different methodologies for evaluating the resilience of neural network systems emerged from the research community. A radiation test is a method that most closely resembles reality. However, these experiments require highly specialized beam radiation equipment that makes them particularly demanding in terms of money and availability. Due to these limitations, software-based fault injection approaches have been consolidated alongside radiation testing as complementary methods to assess the reliability of neural network systems. These methods are based on the emulation of specific fault models in the application-level model of the network. The fault injection approach enables fully controlled experiment campaigns but abstracts from the actual hardware, which may lead to incorrect evaluations. The simulation-based approaches require high and often unaffordable costs in terms of execution time and computational power when dealing with modern and complex architectures. Moreover, they cannot always rely on the actual low-level description of the hardware. Therefore, they are constrained to a higher level of abstraction, such as RTL.

1.1 Main Contributions

The main contribution of this work is to propose a methodology for evaluating the resilience of neural network systems. The method is based on hybrid System-on-Chip (SoC) i.e., platforms combining processor systems and programmable hardware on the same chip. The proposed approach exploits the hybrid platform for emulating the target hardware accelerator and injecting faults in its microarchitectural hardware elements. The reconfigurability feature offered by the platform is used for injecting faults in the hardware configured on the programmable logic by manipulation of the configuration memory data. Hence, the proposed approach involves hardware in the reliability analysis without the demands of a radiation-based approach but working at a lower level of abstraction than traditional software-based approaches. Moreover, it allows to fully control the models and locations of the faults injected in the hardware components, enabling comprehensive and controlled analysis.

The second contribution of this work is the platform for performing the reliability analysis based on this approach. The proposed platform, named *FireNN*, is detailly presented. One of the key features of the platform is to provide the technique and the know-how for modifying the netlist implemented on the programmable hardware through an aware bitstream manipulation, enabling the injection of specific fault models.

Lastly, a reliability analysis of a layer of the AlexNet neural network is performed by using the proposed platform and approach. The results obtained by the fault injection analysis are compared with the results obtained from a software-based fault injection campaign performed at the application level. The application-level fault injection campaign is performed by the software fault injection module embedded in *FireNN* itself, which is independent of the programmable-hardware-based approach.

The proposed approach is presented as a new methodology for reliability analysis to be added to the existing ones. The proposed methodology has advantages and characteristics that differentiate it from the state of the art. The involvement of a hardware platform in the reliability enables the study of the microarchitectural faults, not achievable using software-level simulation approaches that are totally unaware of the underlying hardware. Compared to software simulation of the hardware-level of the hardware platform, this solution offers significant improvement in terms of time analysis, thanks to the use of an actual hardware platform instead of a simulation environment. With respect to radiation tests, the approach provides a means for performing microarchitectural analysis when radiation test is not feasible or as a preliminary analysis in advance of a future radiation test.

This work is organized as follows. Section 2 describes the relevant background for this work on neural networks, programmable hardware, reliability of these technologies, and testing methodologies. Section 3 provides an overview of the related works in the field of neural network reliability, including results, methods, and testing platforms. The proposed approach and the developed platform are presented in sections 4 and 5, respectively. Section 6 is dedicated to the experimental analysis and results. Finally, conclusions are drawn in Section 7.

2 Background

2.1 Neural Networks

Modern neural networks are computing systems made of various cascaded computational layers. Neurons are computational elements typically characterized by weights, an activation function, and a bias. In Fig.1, conceptual schemas of a three-layered neural network (a) and a neuron are illustrated. However, Deep Neural Networks (i.e. networks composed of many layers), and particularly Convolutional Neural Networks (CNNs), have proven to outstand in performance simplest neural networks in applications such as classification, prediction, and object detection. With the increasing of hidden layers (i.e., layers between the input and output layers), the numbers of weights, as well as the required memory and computational power, escalated quickly. The convolutional layers are the most important and computationally intensive among them. Pooling and ReLu are very common operations performed after convolution for downsampling convolution results and introducing non-linearity in the network, respectively.

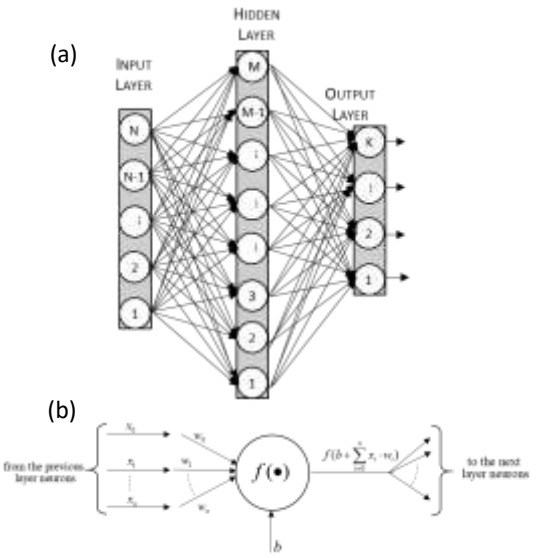

Fig. 1. Conceptual schemas of a three-layer perceptron (a) and a neuron (b).

Modern neural network architectures have hundreds of layers and tens of millions of parameters. The training phase is a complex mathematical process based on algorithms that modify the value of the parameters (e.g., weights and bias) to maximize the accuracy of the network. The huge memory and computational power required by modern architectures made hardware-acceleration essentials during both neural networks training and inference phases.

The choice of the neural network architecture to adopt for performing a specific task is a complex topic. State-of-the-art neural network architectures change quickly, and the adopted design choices can strongly affect the performance and accuracy of the adopted architecture. The high diffusion of neural networks made the born of frameworks necessary in order to easily allow the development and evaluation of neural network architectures, also for a developer with basic or no knowledge of machine learning [15].

2.2 Approaches for Reliability Analyses of NNs

The reliability analyses of neural networks focus primarily on the failures affecting the system during the inference phase since the training phase is performed only once and can be assumed to occur in a controlled environment. Faults can occur in hardware components due to several reasons, such as physical manufacturing defects or radiation effects. When this happens, they can be masked by other components and operations or propagate to the system's outputs, causing errors and eventual failure of the whole system. The state-of-the-art approaches for studying the reliability of neural networks are based on different levels of abstraction from the hardware.

Radiation testing is the method closer to resembling the real case scenario. The actual hardware is exposed to a flux of particles for imitating harsh environments or long periods of execution. Hence, radiation testing requires highly specialized equipment and presents some limit of visibility and control (i.e., more components are usually irradiated during a single radiation test) that restrict its use, especially in the early development phase of a project, when the actual hardware architecture is not finalized yet. When radiation testing is not affordable, different methodologies are used for emulating the system and the faults in order to perform reliability analysis.

Software-level simulation is the method having the highest level of abstraction from the hardware. It relies on an application-level analysis, unaware of the underlying hardware actually implementing the neural network, where faults are emulated in the software model or algorithms of the network. This approach has huge benefits such as high controllability and inherent simplicity and flexibility due to the software level on which it operates. However, the complete abstraction from the actual hardware is partially a limitation. Some faults, such as bit flips in memory cells, are easy to emulate in the software-level model. Differently, errors in hardware microarchitectural components (e.g. impacting the communication interfaces, interconnection lines, timing, specific computational units, and many other elements strictly related to the accelerator hardware architecture), are much more difficult to emulate.

When a more accurate analysis is required**, a hardware-level simulation** relying on a model of the hardware platform must be performed. Faults are injected in the simulated hardware architecture, usually RT- or Gate- level. The low-level simulation allows representing the behavior of the involved hardware providing a more representative analysis. The main disadvantage of this approach is the huge amount of time required for carrying out hardware simulations of complex systems, such as hardware accelerators or microprocessors.

2.3 Programmable Hardware

Modern programmable hardware devices, such as hybrid systems (e.g., All Programmable SoC or AP-SoC) and Field Programmable Gate Arrays (FPGAs), are integrated circuits consisting of a large number of programmable logic blocks and configurable interconnections. FPGAs have been consolidated as hardware accelerators for neural networks thanks to the high performance and flexibility of these devices. AP-SoC extends the traditional FPGAs paradigm integrating one or more processor systems in the same chip. Due to the flexibility offered by the software, AP-SoCs have proven to be extremely suitable for implementing complex applications, including neural networks. Two of the characteristics that have contributed to the success of these technologies are hardware programmability and reconfigurability. The application layer of hardware-programmable devices consists of various types of configurable elements, such as LUTs, DSPs, BRAMs, Flip-Flops, interconnections, and others. The content of the configuration memory, which is the main part of the configuration layer of the device, defines the working functionality of the configurable resources and how they interconnect. For instance, the configuration memory defines the truth table used by the LUTs, the way they are connected to build complex combinational netlists, the configuration of the BRAMs, the operation mode of DSPs, and so on. In particular, the interconnections among blocks are managed

through interconnection matrices. Interconnection matrices consist of Programmable Interconnection Points (PIPs). These interconnections, usually implemented by pass-transistors, are enabled by the configuration data to connect the resources of the application layer for implementing the circuit on the programmable hardware. The configuration memory is written using the configuration data, usually referred to it as the bitstream. The configuration bitstream is generated using vendor tools for a specific device. Reconfigurable hardware can be reconfigured by writing new content in the configuration memory that modifies the implemented netlist.

One of the main limitations of programmable devices is the high hardware expertise required to develop a highly optimized design for this technology. However, modern High-Level Synthesis (HLS) tools are rapidly easing the development for programmable logic, improving obtained performance, and time-to-design. Additionally, new AP-SoC platforms are easing and increasing the use of programmable hardware in hardware-accelerated systems for neural networks. In hybrid platforms (e.g., Zynq AP-SoC), the programmable logic integrated with a processor system on the same chip allows limiting the use of the hardware to the computationally demanding operations, leaving more complex tasks to the processors, combining software flexibility and hardware performance.

3 Related Works

The growing demand for highly reliable neural networks for mission-critical applications and the success of complex neural networks architectures, together with the shrinking of the device technology that makes their hardware-accelerators more brittle, have encouraged research works on reliability evaluation of neural network systems and consequently, methodologies, approaches, and platform for reliability analysis.

In [16], the authors propose a quantitative reliability analysis of three neural network architectures, such as fully connected, CNNs, and Gated Recurrent Units (GRUs). The paper introduces *Ares*, a lightweight framework for empirical analysis based on a software-based approach working at the algorithmic level. The framework allows performing preliminary analyses to obtain insight into where to conduct more detailed fault analysis at the microarchitectural level. The available fault locations are limited to the memory domain and include weights, activities, and hidden states, while data paths are not supported yet. It can inject both static and transient faults. In [17], the authors perform a software-level fault injection, abstracting by the actual hardware, using a fault injection environment based on the darknet framework. However, in [18], the authors report the limitations of a hardware-unaware approach compared to a comprehensive analysis involving the hardware level. In order to carry out fault injections taking into account the hardware platform, they proposed an environment to speed up RTL simulations of DNNs inferences through a multilevel approach based on the pipeline paradigm. Another approach is presented in [19], where an application-level fault injection environment is built on the Tiny-CNN framework for evaluating hardware-accelerated systems. The approach is able to inject in the data path and buffer without considering combinational logic and control logic units. Since the unavailability of RTL implementation of the accelerators, the authors mapped each line of code in the software simulator to the respective hardware element to assess the contribution of each fault injection location in terms of hardware microarchitecture. Also in [20] and [21], the fault injectors rely on the application level but are built on top of a torch-based framework. Most of the works which are not based on application-level rely on radiation testing. In particular, [22] analyzes the reliability of Kepler, Maxwell, and Pascal GPU architectures by NVIDIA. The analysis involves YOLO, Faster R-CNN, and ResNet and it is carried out both through architecture-level fault injection, using NVIDIA SASSIFI fault injector, and radiation testing.

Several works are dedicated to the evaluation of the reliability of circuits implemented on programmable hardware. Since the flexibility offered by programmable hardware, different hardware applications, and cores have been evaluated such as soft processors [23], AXI modules [24], and communication modules [25]. Focusing on the neural network topic, many works are dedicated to the reliability evaluation of neural network applications implemented using a programmable-hardware-based accelerator. In [26], the authors provide a detailed reliability analysis of a neural network accelerator implemented on FPGA. The fault emulation is based on a fault injector module implemented along with the same FPGA with the design under test. Hence, the approach required some minor hardware modifications in order to integrate the fault injection mechanism in the neural network circuit. In [28], the reliability of an AP-SoC platform running a CNN for traffic sign recognition has been tested both through radiation testing and emulating SEUs in the configuration memory of the programmable logic. In [24], a reliability analysis of an FPGA device implementing a CNN is executed. The injection methodology consists of flipping bits in the configuration memory of the device. The work highlights how the network under the test is reliable against SEUs affecting storage bits while SEU affecting bits related to other resources such as DSPs, LUTs, or interconnections are leading to wrong object classifications.

Most of the listed works rely on application-level software-based fault injection approaches, unaware of the underlying hardware platform. Some of the works performing on-hardware evaluation rely on irradiation as a testing methodology. Programmable hardware is used only as the hardware accelerator to be tested, both through irradiation and fault injection. However, in these works, configuration memory manipulation is performed unaware of the resources it will corrupt and the effects on the implemented netlist. In particular, there is no reference either to the possibility to exploit programmable hardware to perform detailed analysis at the microarchitectural level, detect the effects caused by injecting specific fault models and identify the more critical elements of the hardware architecture, which is instead achievable using the methodology we propose.

4 A Hybrid Approach to Reliability Analysis

The proposed method is a general-purpose approach for the reliability evaluation of any kind of neural network system implemented on hardware devices.

The core of the approach is the use of the all-programmable hybrid devices (i.e. system-on-chips combining software and hardware programmability) to enable analysis involving the hardware level. The hardware programmability of the hybrid platforms fulfills two roles: the first, it emulates the hardware architecture of the hardware accelerator running the neural network; the latter, it offers the mechanism for the injection of hardware faults exploiting the reconfigurability feature of the device.

On one hand, the fault injection process is performed through configuration memory manipulation injecting faults directly at the hardware level. On the other hand, the software programmability is managing the fault injection process by applying the test set, performing the evaluation, and the analysis of the fault injection results.

The flow of the approach is illustrated in Fig. 2. The development of the design emulating the target hardware accelerator is performed by using the traditional design flow for programmable hardware. In particular, both HDL and HLS descriptions can be used as a starting point. As an alternative, also third parties developed IPs can be used. The experiment manager module is instrumented with the neural network architecture to be analyzed and data about the experimental analysis to be performed. Hence, the manager handles the integration of the emulated accelerators in the network model and controls the fault injector for performing the experiments and the collection of the results.

The use of the programmable logic of the hybrid devices allows evaluating the effects of faults directly on the hardware. Moreover, it offers a high level of control and visibility due to the possibility to inject faults in specific

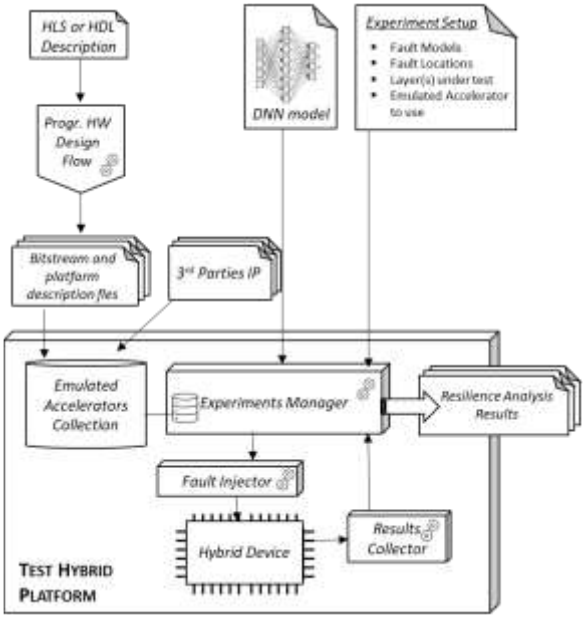

Fig. 2. The conceptual schema of the elements and processes constituting the proposed approach.

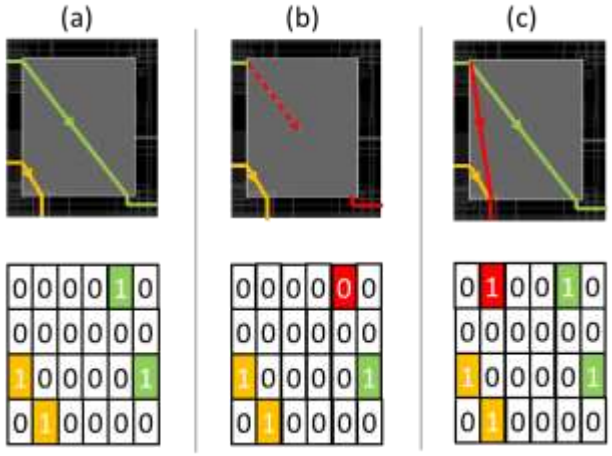

Fig. 3. The implemented netlist without faults (a) can be modified by manipulating the configuration memory to emulate fault models such as open (b) or conflicting (c) interconnections.

hardware components and resources. An additional characteristic is the possibility to perform emulation-based analysis of hardware not yet produced, providing important resiliency information even during the design process of neural network architecture and hardware accelerators (ASICs and FPGA mainly). In particular, when the selected solution for hardware acceleration is programmable logic itself, the actual hardware accelerator is implemented on the programmable hardware without the need to emulate the target accelerator.

Furthermore, the developed platform allows manipulating the configuration data of the programmable hardware for forcing fault models in the resources and interconnections. Even if the tools provided by the vendors prevent the generation of faulty configurations (e.g., open or conflicting nets), it is possible, by bitstream manipulation, to produce these effects in the netlist implemented on the hardware for emulating fault models such as stuck-ats, conflicts, couplings, and others. This concept is illustrated in Fig. 3. The injection of each specific fault model is performed through the modification of particular bits programming a resource. However, the particular role of each specific bit of the bitstream (i.e., which resource they program and how) is not provided by the vendors. Therefore, several projects and research works proved to be able to manipulate the bitstream to modify the implemented netlist without recurring to the vendor tools [30]–[34].

The usability of this approach is highly dependent on the ease of customizing the neural network architecture as well as on the ease of emulating faults that usually require a high level of expertise and knowledge about the manipulation of the configuration memory and architecture of programmable hardware.

5 The FireNN Platform

FireNN is the first platform for enabling resiliency analysis of neural networks both at the application level and at the hardware level via emulation. We selected the Zynq family as the target hybrid platform. In particular, we chose the Zynq-7020 SoC but the implementation is easily extendable to other products of the same family. This AP-SoC is

equipped with two ARM dual-core Cortex-A9 processors and a programmable logic implemented with a 28 nm manufacturing process [29]. The characteristics of the programmable hardware are summarized in Table I.

The platform is based on a two-environment architecture and supports analyses at layer granularity. The support to bottom-end hardware enables to perform parallel analysis exploiting a cluster of testing platforms for reducing the evaluation time with affordable cost and increasing dissemination of the proposed approach. Please note that the first version of FireNN has been presented in [30]. A detailed description of the extended version of FireNN, implementing more features as well as a rearranged structure is presented in this section.

TABLE I
CHARACTERISTICS OF ZYNQ 7020 PROGRAMMABLE LOGIC

Resources	Quantity
Look-Up Tables (LUTs)	53,200
Flip-Flops	106,400
Block RAM	4.9 Mb (140 Blocks)
DSP Slices	220
Configuration Memory	10,008 frames of 3232 bits

5.1 FireNN Architecture Overview

The *FireNN* platform consists of two environments: the *FireNN Machine* and the *FireNN Engine*. Fig. 4 shows a conceptual view of the modules and frameworks involved in the *FireNN* platform. The *Machine* provides the means for defining and modifying the neural network model by extending the PyTorch framework [34]. It runs on the host computer and is the controller of the experimental flow. The *Machine* can communicate with the *FireNN Engine* running on the Zynq platform for triggering the deployment of the emulated accelerator, as well as the fault injection process.

The main purpose of running the *Machine* on the host computer is to offload the computations demand of the whole network from the Zynq in favor of a higher-performance system. Nevertheless, the *Machine* and the *Engine* can also run as two processes on the same hybrid device if its performance is high enough. The *FireNN Machine* provides the APIs for moving the computation performed by a PyTorch module (i.e., one or more software layers) to the programmable logic. The relocation APIs offered by the *Machine* mimics the interface offered by PyTorch to move the computation from the CPU to the GPUs. Practically, this is achieved by instantiating an appropriate hardware-implemented layer on the programmable logic for performing the same mathematical operation that should be performed by the software layer. The hardware layer emulates or implements the target hardware to evaluate. The flow for developing the layer is the same as for traditional hardware accelerators implementable on programmable hardware. This allows using also HDL descriptions for ASICs as a starting point, as well as modern HLS approaches. In particular, when the target is an FPGA-based hardware accelerator, the emulating layer can match almost completely (depending on the specific devices

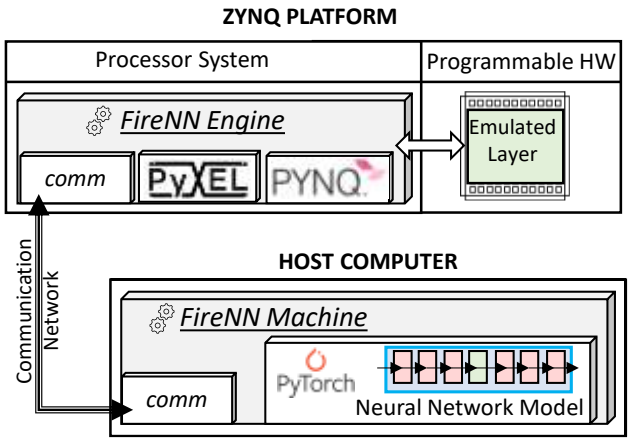

Fig. 4. An architectural view of the FireNN platform.

involved) to the target implementation. The *Engine* runs on the processor system of the Zynq and manages all the operations involving the programmable hardware relying on the Pynq open-source project for managing the programmable hardware from the processor system [35]. The complex process of modifying the configuration memory for inducing specific fault models targeting precise resources is managed by the *Engine* and implemented relying on the PyXEL framework [31].

5.2 Communication Mechanism

The communication between *Machine* and *Engine* is managed by the *FireNN Comm* module. The module enables inter-process communication across the network between the *Machine* and the *Engines* exploiting the python TCP sockets mechanism. Communication is based on messages. The message structure is shown in Fig. 5. In detail, the messages consist of three fields, a header, a JSON header, and a payload. The two headers are mandatory, while the payload is optional. The header is the only field with a fixed length (4 bytes). The JSON header field has a variable length defined in the 4-bytes header. It consists of a serialized JSON dictionary with three mandatory entries. The first entry provides information on the type of message. The second entry is a dictionary with the options and properties related to the specific message type. The third entry contains a second dictionary providing additional information and property for the message processing task of the *Engine*. If the defined protocol allows it for the specific message type, a fourth entry is present to report the presence of the optional payload and its length. This payload is used for transferring weights during the instantiation of the emulated accelerator, as well as input and output data. The communication mechanism has been developed to be

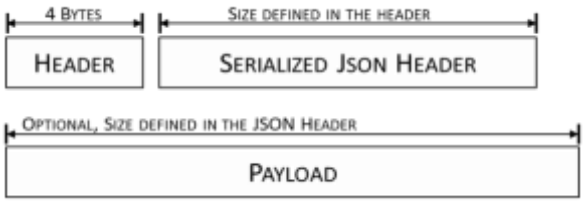

Fig. 5. Structure of the messages used by the communication mechanism.

as transparent as possible to the user. In this way, it is not required for the user to be concerned about the communication protocol between the *Machine* and the *Engine* modules.

5.3 The FireNN Machine and Shells

The *FireNN Machine* manages the neural network architecture and experimental execution. For the description and implementation of the neural network models, the *Machine* relies on the PyTorch framework [34]. The *Machine* extends the PyTorch model of the network providing the mechanism for replacing a PyTorch module with an analogous hardware module running on the programmable hardware.

The instantiation and the management of a specific hardware module on the programmable hardware side are managed by the *Engine*. On the *Machine* side, the relocation is achieved through the encapsulation of one or more neural network layers in a container, named *Shell,* deriving from the module class of PyTorch. A conceptual schema showing how the *Shell* is integrated into the neural network model is illustrated in Fig.6. The relocation API scans the structure of the neural network model and inserts the *Shell* in the place of the original module directly in the topology of the neural network software model. The *Shell* encapsulates the original layer and implements the mechanism for switching the execution from the original module to the hardware module and back. Hence, the *Shell* can interact with the *Engine* to request the computation to be executed by the hardware module. When a *Shell* is instantiated on the *Machine* side, an associated object is instantiated by the *Engine* in its domain. This object is referred to as a *Gear*. The *Gear* is a hardware implementable module emulating the hardware accelerator to analyze. *Gears* and their characteristics are addressed in the next subsection, dedicated to the *Engine* and *Gears*.

The *Shell* is equipped with a routine for automatically inferring the dimensions of input and output data. Indeed, some PyTorch modules are agnostic of I/O data dimensions (i.e, they can receive tensor of any size as input) but their I/O data dimensions are defined univocally by the architecture of the network (e.g., by modules working on fixed tensor sizes such as fully-connected). Hence, the information on input and output dimensionality is needed for the creation of the analogous hardware module.

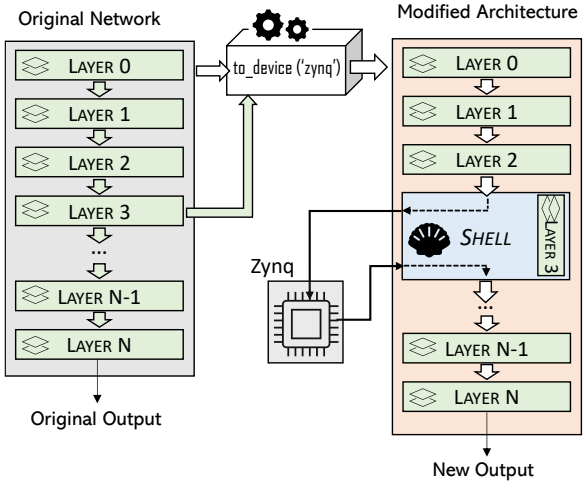

Fig. 6. Conceptual Schema of the integration of the *Shell* in the original model of the network.

The *Shell* has methods to easily switch between the original data path using the PyTorch original layer and the new data path using the hardware module implemented on the Zynq. This feature allows the execution of the inference using the original network or the relocated version, i.e. with a part implemented hardware.

The deployment and execution of an emulated hardware accelerator on the programmable hardware, as well as the emulation of the faults, are all performed on the *Engine* side. However, they are triggered by the *Machine* through the *Shell* since the *Gear* is not directly accessible by the user. Differently, the *Machine* locally implements the methods for executing the injection of different kinds of fault models at the software level since that is performed without involving the hardware emulation.

The *Shell* is also equipped with a tracking mechanism capable of storing input and output data of the *Shell* itself to enable post hoc analysis.

5.4 The FireNN Engine and Gears

The *Engine* runs on the processor system of the Zynq and manages the programmable hardware and the fault injection process. For controlling the programmable hardware, the *Engine* relies on the PYNQ project for Zynq. PYNQ is an open-source project, supported by Xilinx to ease the exploit of the programmable hardware through a python framework [35]. It provides the APIs for configuring the programmable hardware of the Zynq, as well as the APIs for managing I/O and communication between the programmable hardware and processor system. The *Engine's* main tasks are the deployment, execution, and fault injection of the *Gears.* When a *Shell* is instantiated on the *Machine* side, an associated *Gear* is instantiated by the *Engine.*

A *Gear* represents a computational module that can be implemented on the programmable hardware. It emulates a target hardware accelerator. A *Gear* consists of three elements: an interface, a driver, and an implementation file (i.e., a bitstream). A conceptual schema summarizing the element composing the *Gear*s is reported in Fig.7. The interface is common to all the *Gear*. It allows the *Engine* to manage the Gear independently of their particular hardware implementation. The driver provides the APIs for using the specific hardware module through the interface. The implementation file is the bitstream containing the configuration data for programming the programmable hardware with the hardware module. If necessary, when a *Gear* object is created, the characterizing parameters of the original neural network layer it is replacing (e.g. weights, bias, etc.) need to be transmitted by the *Shell* during the relocation process.

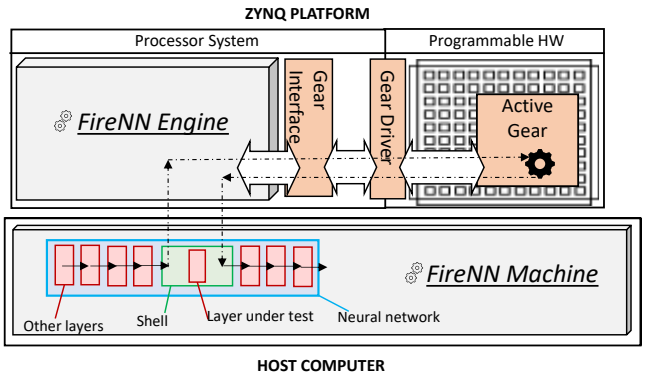

Fig. 7. Architectural Schema of the elements composing a Gear.

When on the *Machine* side a *Shell* requires the execution of a computation on a specific *Gear*, the *Engine* configures the programmable hardware with the specific *Gear* implementation, receives the input data, triggers the computation on the hardware, and transmits back the results. Hence, each *Gear* is associated with a specific computational layer on the software model of the network and emulates a particular hardware accelerator. The development of a *Gear* is performed using the traditional development flow for programmable hardware (e.g., starting by HDL or HLS description), or it can be based on IPs by third parties.

The fault injection is a complex process performed by the *Engine* relying on the PyXEL framework for performing the bitstream manipulation [31]. It can modify the bits of the configuration bitstream for changing the configuration of specific resources of the programmable hardware. In particular, PyXEL makes it possible to manipulate those bits for injecting a specific fault model, such as soft error affecting the truth table of a LUT, or the open-interconnection fault model (by disabling a programmable interconnection as already shown in Fig. 3). The *Gear* common interface is provided with a tunable timeout mechanism to avoid endless waits as a result of faults injected during the injection operation. When the injection of a specific fault model is requested by the *Shell* a faulty version of the *Gear* is created and used for the computation. The *Engine* can be instructed to inject faults in a specific entity (e.g., the AXI Interface), in specific resources (e.g., BRAMs interconnections, LUTs truth tables), or induce specific faults (e.g., open interconnections).

6 Experimental Setup, Analysis, and Results

The experimental evaluation of the *FireNN* platform has been applied to the reliability analysis of the AlexNet neural network. AlexNet has been implemented considering the PyTorch model provided by *torchvision* [36]. The 2D-convolution computation within the fifth convolutional layer of AlexNet has been implemented as a hardware accelerator using Vivado HLS [37]. We performed different fault injection campaigns on the specific convolutional layer of the same neural network model. Performed analyses include both traditional software-level analysis and analyses based on the proposed approach relying on the emulation of accelerators and fault models using hybrid devices. *FireNN* allows to measure several parameters including error rates, failure rates, ratios between failures and errors, timeout events, and distribution of degradations and misclassifications for the evaluation set.

6.1 Neural Network Model

The architecture of the network used in the experimental analysis is presented in Fig. 8. The network model is the version of the AlexNet network provided by *torchvision*. AlexNet is a CNN for object classification [38]. The model is provided trained using the ImageNet dataset as the training set [39]. The trained model of the network is able to classify 1,000 different classes. The architecture of AlexNet consists of several layers implementing convolution, pooling, and ReLu operations.

The input is a tensor of dimensions 3×224×224 representing an RGB picture. When a specific implementation for the hardware-accelerated version of a layer is provided, the platform can carry out reliability analysis of a neural network model with a layer granularity. The layer selected for the reliability analysis is the last convolutional layer of the network since in [19] it has been identified as the most sensitive one of the convolutional layers. The layer is characterized by 590,080 parameters (also referred to as weights) consisting of 256 bias and 256 kernels, each one has dimensions of 256×3×3.

The output of the last fully connected layer of the neural network is an array of values, each one associated with a class. This value assesses the probability of the input belonging to the specific class. Using a normalized exponential function (usually referred to as softmax), these values are normalized and reduced to a probabilistic distribution over all the labels. Hence, the final output of the neural network is a rank of labels with an associated percentage of confidence to each item. The item with the higher confidence represents the classification output.

6.2 Evaluation Set and Error Classification

The inputs of the neural network are images representing objects belonging to the set of the 1,000 labels used in the training phase. The images need to be preprocessed (e.g., cropped and normalized) in order to be suitable as inputs

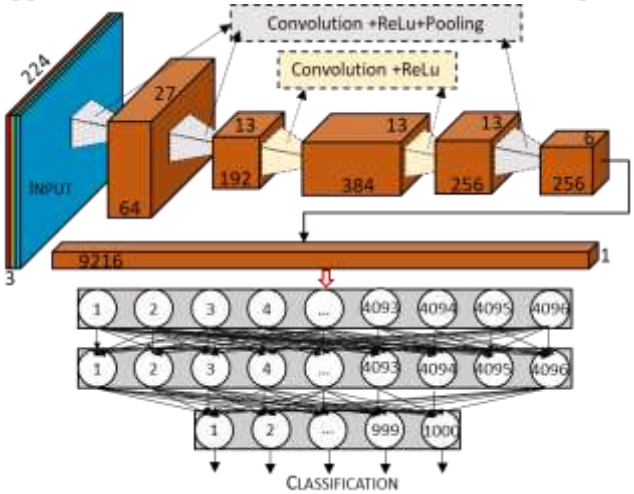

Fig. 8. Schema of the AlexNet model used in the experiments.

of the network. Since our goal is to assess the reliability instead of the accuracy of the network, we are interested in the deviation from the unfaulty behavior independently of the original accuracy of the network. Therefore, since not strategic for our purpose, we picked 50 pictures from the ImageNet collection on fauna [39] as evluation set. The main objective of the reliability analysis is to identify if an injected fault will produce a modification in the confidence associated with each label, possibly causing a change with respect to the original classification. Hence, the original model is initially used for obtaining the results of an unfaulty run. Then, the golden results are used for detecting errors through comparison with the output obtained by a fault-injected model of the network. The errors detected by this procedure are classified into three groups.

1. *Misclassification*: the label with higher confidence has changed due to the injected fault.
2. *Degradation*: the confidences of one or more labels have changed due to the injected fault.
3. *Timeout*: the injected fault prevents the network from completing the classification.

Since the misclassification is usually caused by a severe degradation of the confidence values and it totally changes the expected results, we considered it as a critical failure instead of an error only potentially affecting the classification. The failure rate is computed as the number of injections that caused misclassification out of the total while the error rate is computed as the number of injections that caused at least one degradation (independently if they do or do not lead to misclassification) out of the total.

Since the evaluation set is composed of different images with heterogeneous characteristics (e.g., producing classification with different confidence values), it could be possible to have a mixed outcome as a result of fault injection (i.e., a mix of misclassifications, errors, and correct results assigned to different inputs of the evaluation set). If at least one input has been misclassified the error is categorized as misclassification. Degradations follow the same rule. Additionally, we considered a deviation minor than $1{\cdot}10^{-4}$ in the confidence percentage as negligible since they can be caused by the use of the IEEE-754 standard for data representation and computation. Hence, a deviation of less than $1{\cdot}10^{-4}$ percentage points is not classified as an error.

6.3 Fault Models

In the reliability analyses carried out in this section, we evaluated three different fault models. Single Event Upset is a well-known phenomenon caused by the interaction between the hardware and ionizing particles and affecting memory cells during the execution of the application. SEUs are very common in space but they can affect also systems working at sea level.

Subsection 6.5 report a reliability evaluation against SEUs affecting the weights and the inputs (i.e., the memory cells storing the values) of the software-level simulation of the analyzed neural network. Errors are emulated directly modifying the value of the variables at the bit-level. Errors are injected in the value of weights and layer inputs during runtime.

Even if caused by the same phenomenon, SEUs affecting the configuration memory of programmable hardware have to be considered differently. Due to the characteristics of programmable hardware, SEUs affecting the configuration memory will result in a permanent fault affecting the hardware architecture of the circuit implemented on the programmable hardware (until the hardware is not programmed with a new circuit). In subsection 6.6, the reliability analysis of a circuit implementing a convolutional layer of the AlexNet network is performed. SEUs in configuration memory have been modeled as bit flips affecting the content of the memory. The fault has been emulated in the configuration memory of the programmable hardware programming the board with a faulty bitstream and the implemented circuit is used for executing the convolutional layer under test. These SEUs in configuration memory can cause various actual faults in the circuit implemented in the hardware according to which bit (randomly selected) has been corrupted. For example, if a memory cell programming a LUT is affected, it may cause a logical fault while if the memory cell is related to a programmable interconnection it may cause an open fault or an antenna

Finally, section 6.7 has been dedicated to evaluating the effects of open faults exclusively. Open faults can be caused by several causes, such as fabrication defects, aging effects. In programmable hardware, they can also derive by undesired modification of the configuration memory caused by SEUs. The open fault model has been emulated in the architecture of the hardware accelerator implemented on the programmable hardware by an aware manipulation of the configuration memory bits programming the nets of the implemented circuit. The fault model has been emulated in the hardware programming the device with a faulty bitstream.

6.4 Hardware Accelerator

As a hardware accelerator, we developed a hardware module implementing the fifth convolutional layer of the AlexNet network. The hardware accelerator has been developed using Vivado HLS. It computes 2-D multichannel convolution between inputs with dimension 13×13 and a 3×3 kernel and it has 256 input channels and 256 output channels. Data are represented with a 32-bits floating-point representation, accordingly with the data type used by the PyTorch model of the overall network. Data transfer between the processor system and programmable hardware is performed by the FPGA direct memory access to transfer streams of data from the DDR memory to the programmable logic and vice versa. The IP Core is pipelined and performs convolution using an algorithm based on

TABLE II
RESOURCES UTILIZATION OF HARDWARE CONVOLUTIONAL IP

Resources	AXI Modules	Core	Total
Slice LUTs	4,452	31,746	36,198
Slice Registers	5,795	25,476	31,271
Block RAMs	16	67	83
DSP Slices	0	48	48
Muxes	25	87	102

two buffers and a shifting window. The interface of the core is implemented with an AXI4-Lite control register interface. The resources used by the convolutional core and the communication modules (i.e., AXI modules) are reported in Table II.

6.5 Software-based Reliability Analysis

We used software-based fault injection campaigns to achieve two reliability analyses considering faults affecting weight and bias of the layer, and faults affecting input and output data of the layer.

In the first fault injection campaign, we used a traditional approach based on software fault injection in order to perform further results comparison. We emulated SEUs (i.e., a single bit flip) in the 32-bits floating-point representations of the parameters (weights and bias) of the fifth convolutional layer of the AlexNet model. Therefore, each faulty network presents a single bit flip in its parameters with respect to the unfaulty version. The fault injection campaigns have been executed using the software-level fault injector embedded in the *FireNN* platform.

We performed 10,000 experiments and we run the classification task on the whole evaluation set under the same faulty bit in each experiment. The fault locations (i.e., parameter and bit to inject) have been randomly generated for each experiment. As a result, we obtained an error rate of 40.57% and a failure rate of 2.10%, while 5.18% of the detected degradations led to misclassification. We have not observed any timeout errors.

As we explained in subsection 6.2, misclassification of a single input image of the evaluation set is enough to categorize a fault as causing misclassification. The impact of a fault considering the overall evaluation set can be different since a fault may affect from 1 to all the images of the evaluation set. Therefore, we analyzed the distribution of the faults leading to misclassification over the number of outputs for which the misclassification has been observed and we reported the data in Fig. 9.a. Fig. 9.b reports the data for degradations. This result suggests that it is common for faulty networks presenting misclassifications to have two possible behaviors. The first one is to have misclassification on very few outputs. This could happen due to specific elements of the evaluation set being hard to classify. Therefore, they are more sensitive than others to variations induced by faults and they will cause a misclassification also with small deviations from the unfaulty behavior. The second one is to affect a very high percentage of the evaluation set. This effect could be caused by the stimulation of highly critical bits of the parameters of the neural network layer.

A second fault injection campaign has been performed using the same fault model but affecting the inputs or outputs of the module instead of the parameters. The fault is injected in the same location for all the inputs of the module (i.e, on the same data of the input or output tensor and the same bit of the data) for each experiment.

We performed 10,000 experiments and evaluated the effect of each single bit flip singularly on the whole evaluation set consisting of 50 pictures. We obtained an error rate of 46.16% and a failure rate of 15.48% with a ratio of failures to errors of 33.53%. We have not observed any timeout errors. The distribution of the faults is illustrated in Figure 10.a for the misclassification and Figure 10.b in the case of degradation. The result shows how the probability to detect multiple errors and failures decreases both for misclassifications and degradations with the increasing of multiplicity. This is a reasonable behavior if we consider that a specific feature extracted by the current and previous layers can play a critical role in the classification of a specific input or class but to be less important for others.

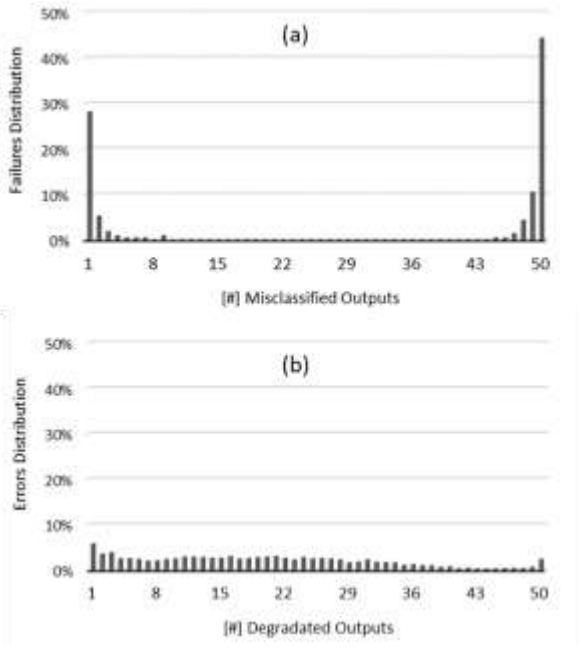

Fig. 9. Distribution of the misclassification (a) and degradation (b) categories (resulting from fault injection in the parameters) over the number of outputs experiencing the effect.

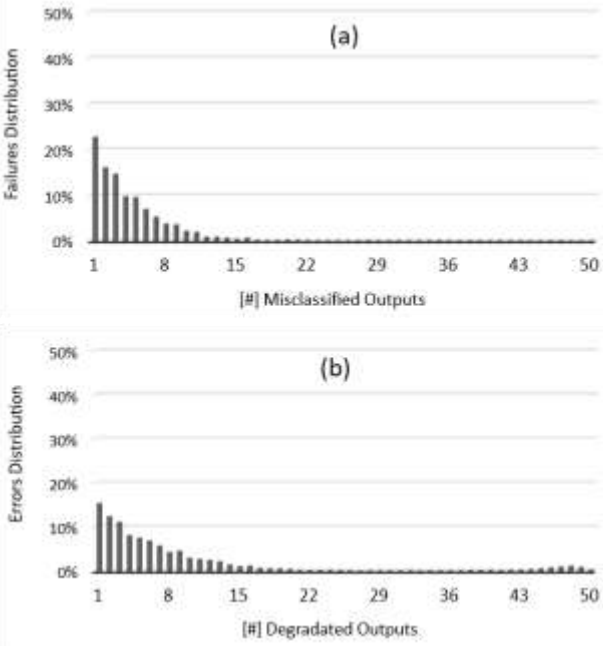

Fig. 10. Distribution of the misclassification (a) and degradation (b) categories (resulting from fault injection in the data) over the number of outputs experiencing the effect.

6.6 Hybrid-based Reliability Analysis against SEUs

The hybrid fault injection campaign allows the analysis of the behavior of the platform and addresses the reliability of the hardware accelerator when it is physically implemented on programmable hardware. In detail, we emulate SEUs in the configuration memory of a Zynq device in order to mimic hardware faults affecting the structure of the neural network implemented on the programmable logic. In the considered case, the hardware design implemented on the programmable logic is the convolutional module already presented in subsection 6.4. The hardware core is used as the fifth convolutional layer of the AlexNet model (presented in subsection 6.1). The current fault injection campaign targets a hardware-accelerated version of the software convolutional layer analyzed by the previous software-based reliability analysis.

SEUs affecting configuration memory are a well-known source of errors for hardware accelerators implemented on SRAM-based programmable hardware.

We would like to emphasize that the injection of faults in the configuration memory of the hardware-programmable devices can cause undesired modification in the structure of the implemented circuit, affecting not only data but also data path, computational elements, interconnections, etc. Hence, due to the relation between configuration memory and circuit configuration, soft errors in the configuration memory produce a modification in the hardware of the circuit until the device will be reconfigured [40]–[42].

We injected 10,000 single bitflips in the configuration memory of the device implementing the hardware accelerator. We run the classification of the whole evaluation set consisting of 50 pictures for each faulty configuration of the hardware-accelerated network. Please notice that, due to the intrinsic characteristics of programmable devices, not all the bitflips in configuration memory will generate faults in the implemented netlist. To elaborate more, since only a subset of resources is used by the implemented netlist, injections could target unused resources.

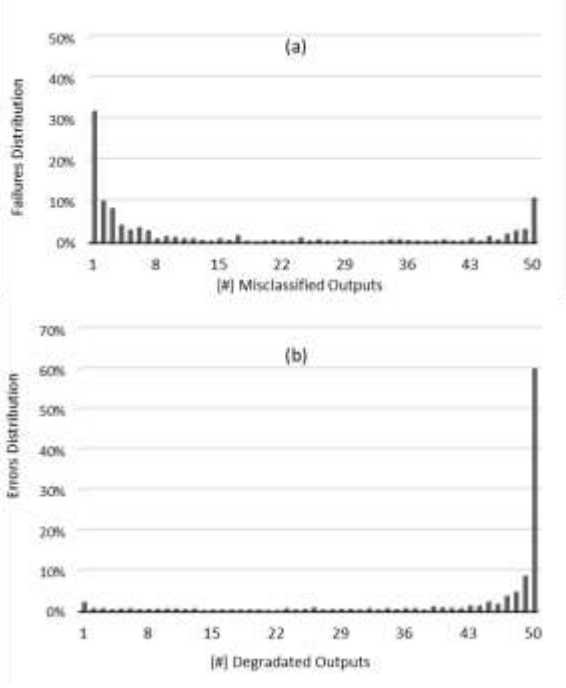

Fig. 11. Distribution of the misclassification (a) and degradation (b) categories (resulting from SEU fault model injection) over the number of outputs experiencing the effect.

As a result, we obtained an error rate of 11.05%, a failure rate of 5.12%, and a ratio between failures and errors of 46.33%. Besides, we experienced a timeout in 0.40% of the injections. Fig. 11 reports the detailed distribution for misclassification and degradation as done in the previous fault injection analyses.

We obtained insight into the location randomly selected for the injection and the error and failure rates regarding them applying the PyXEL framework embedded in *FireNN*. The achieved results are reported in Table III, where the column Hits reports how many injections hit one of the resources (used or unused) listed in the resource column, while the Hits (Used) column reports the number of injections that hit a used resource. The third and fourth columns list the error rate and the failure rate generated by injection of a used resource among the listed ones, respectively. The row of the table referred to as *empty* collects the injections which targeted particular sections of the Zynq configuration memory not configuring any resource.

The error and failure rate associated with a specific resource is valuable information provided by the use of *FireNN*, concerning the hardware domain. Therefore, it cannot be inferred using traditional software-based fault injection approaches. From the analysis, we have been able to identify that the routing interconnections are both the most used resource and the ones with the higher sensitivity to SEUs.

TABLE III
RESUME OF THE FAULT INJECTION CAMPAIGN OF SEUS IN CONF.MEMORY

Resources	Hits (Any)	Hits (Used)	Err. Rate (of Used)	Fail. Rate (of Used)
Routing	4,577	3,584	23.94%	11.43%
LUTs	1,370	1,058	10.49%	3.11%
Block RAMs	493	384	13.54%	7.55%
DSP	431	324	14.81%	7.10%
Flip-Flops	365	286	10.38%	4.89%
Empty	2337	-	-	-
Others	427	288	9.85%	9.09%

6.7 Hybrid-based Reliability Analysis against Open-Routing Model

In the current subsection, we present a resiliency analysis carried out using the *FireNN* platform. In the analysis, we emulate the open fault model in the interconnections of a design implemented on programmable hardware, also named the open-routing fault model. Open faults are the most frequent error event happening in programmable logic devices [43]. As a result of the reliability analysis reported in subsection 6.6, we detected interconnections as a critical resource for the accelerator introduced in subsection 6.4. In order to investigate more in detail the issues related to interconnections faults, we randomly injected open-routing faults in the routing of the implemented hardware accelerator. In particular, the platform modified the bits related to a specific interconnection to create an

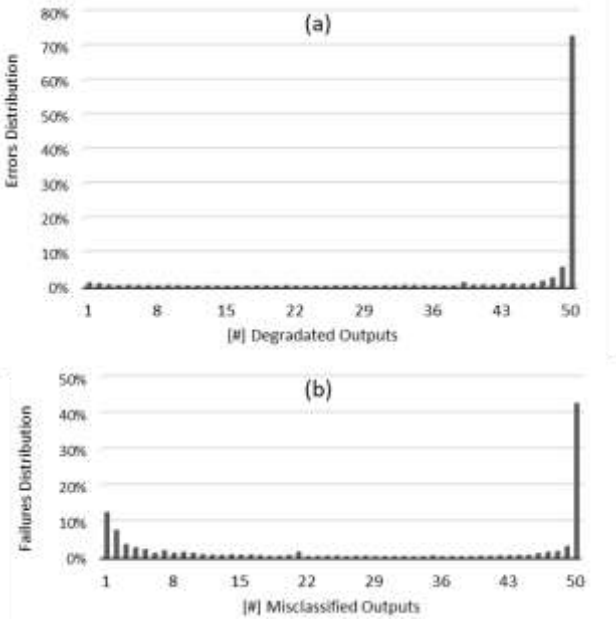

Fig. 12. Distribution of the misclassification (a) and degradation (b) categories (resulting from open-interconnection fault model injection) over the number of outputs experiencing the effect.

open fault in the netlist, similarly to the example reported in Fig. 3.b. This specific analysis is a prove of how *FireNN* can be used for injecting various and specific fault models in the hardware structure of hardware accelerators either emulated or implemented on programmable hardware.

The fault injection campaign consists of 10,000 injections. Interconnections, where to inject the open routing fault model, have been randomly selected among the programmable routing segments used by the design implemented on the hybrid device. This means that, differently from the previous reliability analysis, we are not hitting unused resources. The analysis showed an error rate of 59.62% and a failure rate of 40.07%. The ratio of failures to errors for the current fault injection campaign is 67.21%. 2.78% of the fault injections led to timeout events. Fig.12 shows in detail the distribution of detected events for misclassifications and degradations. From Fig.12.a, it results that errors induced by open-interconnection fault models are very likely to affect all the outputs of the evaluation set. Additionally, the ratio of failures and errors shows how a very high percentage of them led to misclassification. Similarly, Fig. 12.b shows how a very large part of misclassifications induced by open-interconnection faults will affect a very large portion of the outputs.

6.8 Results of the Reliability Analyses

The performed reliability analyses have highlighted the variety of information that is possible to obtain using the proposed method and using the *FireNN* platform. In Table IV, the obtained measures are resumed. The hybrid-based approach allowed us to obtain insight into the resiliency of the hardware implementation and its microarchitectural elements that cannot be provided using a software-based approach. For the analyzed hardware accelerator, we found how the open-routing fault model led to a high degradation of confidence that often produces misclassification, especially compared to SEUs in weights and data or random SEUs in the configuration memory. These analyses have been enabled by the proposed platform, providing the mean for emulating the fault models and integrating the implementation of specific hardware accelerators in the model describing the neural network architecture.

TABLE IV
RESUME OF THE ALEXNET RELIABILITY ANALYSES

Method	Software Fault Injection		Hybrid-based Fault Emulation (*FireNN*)	
Fault Model	SEU in Weights	SEU in Data	SEU in Conf. Memory	Open Routing
Error Rate	40.57%	46.16%	11.05%	59.62%
Failure Rate	2.10%	15.48%	5.12%	40.07%
Fail./Err.	5.18%	33.53%	46.33%	67.21%
Timeouts	0%	0%	0.40%	2.86%

In order to provide additional analysis exploiting the proposed platform, we also evaluated a layer of the ResNet-18 neural network. The evaluation analyzed a hardware implementation of the last convolutional layer of the network architecture. ResNet-18 contains many more layers with respect to the AlexNet network. However, using the hybrid platform the layers preceding the layer under test can be run in software, focusing the analysis on the layer of interest. Since the similarity of the convolutional layers used in the traditional convolutional networks, the convolutional core under analysis is similar to the one exposed in detail for the AlexNet layer. The main difference is the use of 512 channels instead of 256. The results of the evaluations against SEUs in the configuration memory and the Open Routing fault model (analogous to the ones performed in subsections 6.7 and 6.8) are resumed in Table V.

TABLE V
RESUME OF THE RESNET-18 RELIABILITY ANALYSES

Method	Hybrid-based Fault Emulation (*FireNN*)	
Fault Model	SEU in Conf. Memory	Open Routing
Error Rate	12.93%	60.38%
Failure Rate	5.81%	42.17%
Fail./Err.	44.93%	69.84%
Timeouts	0.51%	2.86%

6.9 Comparison with state-of-the-art Techniques

The proposed methodology has shown to be a valid alternative to the traditional techniques typically adopted for analyzing the reliability of neural network systems. The methodology provides a solution for performing reliability analyses aware of the hardware architecture in a short time and without requiring highly specialized equipment.

The software-level simulation analysis is still the fastest method for performing reliability analysis. For instance, the campaigns injecting faults during software-level simulation carried out in this section required less than 10 hours. However, the abstraction from the hardware level prevents the analysis to be comprehensive. The

dependency of the application reliability on the specific hardware platform adopted is well known and the main reason why more demanding techniques, such as hardware-level simulation and radiation testing, are needed [13][14][18][28]. This dissimilarity was also found in the comparison between software-level simulation and hardware-based fault injection reported in the experimental analysis section.

With respect to hardware-level simulations, the main advantage offered by the approach is the speedup of the time needed for the analyses. The hardware-based simulation approach is known to be extremely demanding in terms of computational power and execution time. For instance, the authors of [18] propose a method for speeding up hardware-based simulation. In the work, they report a time of 25 minutes for performing inference of a single input image using a 7 layer CNN simulated at RTL-level using a server equipped with a dual Intel Xeon CPU E5-2680 v3 and 256 GB. The *FireNN* platform runs on a Zynq-7020 board and required an average time of 28 seconds for evaluating 50 input images, including the time for generating the fault and communicating with the host computer. For performing a campaign of 10,000 fault injections with an evaluation set of 50 images the average required time was about 80 hours. Compared with the about 208,333 hours taken by traditional RTL simulation, or with the about 45,833 hours using the method proposed in [18], the use of hardware emulation can guarantee a significant speedup of several orders of magnitude.

Additionally, when larger neural networks or hardware platforms need to be emulated the simulation complexity grows quickly along with execution time and computational demands. The hybrid methodology we are proposing addresses the scalability issue in two different ways. Firstly, the possibility to evaluate on the hardware platform only an architectural layer at once (as we showed in our experimental analysis) enables a layer-level study. Hence, the implementation of only a single or a subset of layers is present on the hardware while the rest of the network can be simulated at the software level. This allows using also small devices for preliminary analysis on large neural networks on the condition that the chosen hardware platform can contain at least one of the layers of the chosen neural network architecture. Secondly, such an analysis scales along with the feasibility of the project. Similar to what happens in a radiation test, scalability is limited by the implementability of the design on the programmable hardware and by the number of faults to analyze.

As already stated, radiation testing is still necessary when a deeply accurate analysis is required. However, the hybrid methodology provides a good tradeoff for performing preliminary microarchitectural analysis involving specific resources or fault models without the need and the costs for highly specialized equipment. This also makes it possible to use it in the early stages of the development process, when the hardware platform has not been firmly chosen, allowing the eventual hardware platform to be modified according to the results obtained.

7 Conclusions

Hardware accelerators are crucial for neural network applications. Hence, the resiliency of these applications is strictly dependent on the hardware architecture and neural network model. In this work, we proposed a new approach based on the emulation of the hardware accelerators and fault models through hybrid devices. The feasibility of the approach has been proven by introducing the first platform for enabling an analysis comprehensives of the hardware level without recurring to radiation testing. We analyzed the resiliency of a convolutional hardware core integrated into a software model of the AlexNet neural network. Using the FireNN platform, we could compare the results obtained by software-based and hybrid-based approaches. FireNN allowed us to emulate particular fault models typical of programmable hardware devices (i.e., SEUs) and ASICs (i.e., open-routing) domains and evaluate the sensitivity of specific resources, providing valuable insight on the resiliency and the critical elements of the system.

References

[1] Y. LeCun, Y. Bengio and G. Hinton, "Deep learning", *Nature*, vol. 521, pp. 436, 05 2015, [online] Available: http://dx.doi.org/10.1038/nature14539.

[2] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, 2016.

[3] L. Deng *et al.*, "Recent advances in deep learning for speech research at Microsoft," 2013.

[4] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. E. Mohamed, and H. Arshad, "State-of-the-art in artificial neural network applications: A survey," *Heliyon*. 2018.

[5] Z. Q. Zhao, P. Zheng, S. T. Xu, and X. Wu, "Object Detection with Deep Learning: A Review," *IEEE Transactions on Neural Networks and Learning Systems*. 2019.

[6] M. Capra, B. Bussolino, A. Marchisio, M. Shafique, G. Masera, and M. Martina, "An Updated Survey of Efficient Hardware Architectures for Accelerating Deep Convolutional Neural Networks," *Future Internet*, vol. 12, no. 7, p. 113, 2020.

[7] Mittal, S. A survey of FPGA-based accelerators for convolutional neural networks. *Neural Computing and Applications 32*, 1109–1139 (2020).

[8] G. Furano *et al.*, "Towards the Use of Artificial Intelligence on the Edge in Space Systems: Challenges and Opportunities," *IEEE Aerospace and Electronic Systems Magazine*, vol. 35, no. 12, pp. 44–56, 2020.

[9] F. Falcini, G. Lami, and A. M. Costanza, "Deep Learning in Automotive Software," *IEEE Software*, 2017.

[10] A. Luckow, M. Cook, N. Ashcraft, E. Weill, E. Djerekarov and B. Vorster, "Deep learning in the automotive industry: Applications and tools," *2016 IEEE International Conference on Big Data (Big Data)*, Washington, DC, 2016, pp. 3759-3768.

[11] ISO, *ISO 26262-2 : Road Vehicles-Functional Safety*. 2018.

[12] RTCA, "Design assurance guidance for airborne electronic hardware," *Do-254*. 2000.

[13] S. Mittal, "A survey on modeling and improving reliability of DNN algorithms and accelerators," *Journal of Systems Architecture*, vol. 104, no. August 2019, p. 101689, 2020.

[14] Y. Ibrahim *et al.*, "Soft errors in DNN accelerators: A comprehensive review", *Microelectronics Reliability*, Volume 115, 2020, ISSN 0026-2714.

[15] A. Parvat, J. Chavan, S. Kadam, S. Dev, and V. Pathak, "A survey of deep-learning frameworks," *2017 International Conference on Inventive Systems*

and Control (ICISC), Coimbatore, 2017, pp. 1-7.

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

[17] A. Ruospo, A. Bosio, A. Ianne and E. Sanchez, "Evaluating Convolutional Neural Networks Reliability depending on their Data Representation," *2020 23rd Euromicro Conference on Digital System Design (DSD)*, Kranj, Slovenia, 2020, pp. 672-679.

[18] A. Ruospo, A. Balaara, A. Bosio and E. Sanchez, "A Pipelined Multi-Level Fault Injector for Deep Neural Networks," *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Frascati, Italy, 2020, pp. 1-6.

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

[20] B. F. Goldstein *et al.*, "Reliability Evaluation of Compressed Deep Learning Models," *2020 IEEE 11th Latin American Symposium on Circuits & Systems (LASCAS)*, San Jose, Costa Rica, 2020, pp. 1-5.

[21] M. A. Neggaz, *et al.*, "A Reliability Study on CNNs for Critical Embedded Systems," *2018 IEEE 36th International Conference on Computer Design (ICCD)*, Orlando, FL, USA, 2018, pp. 476-479.

[22] F. F. d. Santos et al., "Analyzing and Increasing the Reliability of Convolutional Neural Networks on GPUs," in *IEEE Transactions on Reliability*, vol. 68, no. 2, pp. 663-677, June 2019.

[23] L. A. Aranda, et al. "Analysis of the Critical Bits of a RISC-V Processor Implemented in an SRAM-Based FPGA for Space Applications" *Electronics* 9, no. 1: 175, 2020.

[24] C. De Sio, S. Azimi, and Luca Sterpone, "On the Evaluation of SEU Effects on AXI Interconnect Within AP- SoCs", *International Conference on Architecture of Computing Systems*, pp. 215-227, 2020.

[25] Z. Gao, L. Zhang, R. Han, P. Reviriego and Z. Li, "Reliability Evaluation of Turbo Decoders Implemented on SRAM-FPGAs," *2020 IEEE 38th VLSI Test Symposium (VTS)*, 2020, pp. 1-6.

[26] D. Xu et al., "Reliability Evaluation and Analysis of FPGA-Based Neural Network Acceleration System," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 3, pp. 472-484, March 2021.

[27] I. C. Lopes, *et al.*, "Reliability analysis on case-study traffic sign convolutional neural network on APSoC," 2018 IEEE *19th Latin-American Test Symposium (LATS)*, Sao Paulo, 2018.

[28] B. Du, *et al.*, "On the Reliability of Convolutional Neural Network Implementation on SRAM-based FPGA," *2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Noordwijk, Netherlands, 2019, pp. 1-6.

[29] Xilinx, "Zynq-7000 SoC Technical Reference Manual," *Ug585*, 2018.

[30] C. De Sio, S. Azimi, L. Sterpone, "An Emulation Platform for Evaluating the Reliability of Deep Neural Networks," *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Frascati, Italy, 2020, pp. 1-4.

[31] L. Bozzoli, C. De Sio, L. Sterpone and C. Bernardeschi, "PyXEL: An Integrated Environment for the Analysis of Fault Effects in SRAM-Based FPGA Routing," *2018 International Symposium on Rapid System Prototyping (RSP)*, Torino, Italy, 2018, pp. 70-75.

[32] K. Dang Pham, E. Horta and D. Koch, "BITMAN: A tool and API for FPGA bitstream manipulations," Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, Lausanne, 2017, pp. 894-897.

[33] T. Haroldsen, B. Nelson and B. Hutchings, "RapidSmith 2: A Framework for BEL-Level CAD Exploration on Xilinx FPGAs", *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015, pp. 66–69.

[34] A. Paszke, *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library", 2019, ArXiv, abs/1912.01703.

[35] "pynq.io", 2016, [online] Available: www.pynq.io.

[36] S. Marcel, and Y. Rodriguez. "Torchvision the machine-vision package of torch." *ACM Multimedia*, 2010.

[37] Xilinx, "Vivado High-Level Synthesis," *Ug902*, 2018.

[38] A. Krizhevsky *et al.*, "ImageNet Classification with Deep Convolutional Neural Networks Alex," *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, 2012.

[39] J. Deng, W. Dong, R. Socher, L. Li, Kai Li and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," *2009 IEEE Conference on Computer Vision and Pattern Recognition*, Miami, FL, 2009, pp. 248-255.

[40] B. Du *et al.*, "Ultrahigh Energy Heavy Ion Test Beam on Xilinx Kintex-7 SRAM-Based FPGA," *IEEE Transactions on Nuclear Science*, 2019.

[41] C. De Sio, *et al.*, "Radiation-induced Single Event Transient effects during the reconfiguration process of SRAM-based FPGAs," *Microelectronics Reliability*, vol. 100–101.

[42] C. Bernardeschi, *et al.*, "UA2TPG: An untestability analyzer and test pattern generator for SEUs in the configuration memory of SRAM-based FPGAS," *Integration, the VLSI Journal*, 2016.

[43] C. Bernardeschi, L. Cassano, A. Domenici, and L. Sterpone, "ASSESS: A Simulator of Soft Errors in the Configuration Memory of SRAM-Based FPGAs," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 9, pp. 1342-1355, Sept. 2014.

Corrado De Sio received the B.S. and M.S. degrees in Computer Engineering from the University of Pisa, Pisa, Italy, in 2018. He was a Research Assistant in the CAD and Reliability Group, Department of Computer and Control Engineering, Politecnico di Torino. Currently, he is a Ph.D. Student at Politecnico di Torino. His research interests include the reliability of reconfigurable devices, radiation effects, and soft-errors.

Sarah Azimi received her Ph.D. from Politecnico di Torino, Turin, Italy, in 2019. Currently, she is working in the CAD & Reliability group of the Department of Computer and Control Engineering In Politecnico di Torino as an Assistant Professor. Her research interests include fault-tolerant electronic design, physical models and validation platforms, radiation effects on components and systems. She is a member of the IEEE.

Luca Sterpone received the M.S. and Ph.D. degrees in computer engineering from the Politecnico di Torino, Italy, in 2003 and 2007, respectively, where he is currently a Full Professor with the Department of Computer and Control Engineering. He has authored more than 200 papers and he received several awards for his research activities. His current research interests include reconfigurable computing, computer-aided design algorithms, fault tolerance architectures, and radiation effects on components and systems.