Usability-based Cross-Layer Reliability Evaluation of Image Processing Applications

Cristiana Bolchini, Luca Cassano, Andrea Mazzeo, Antonio Miele
Dipartimento di Elettronica, Informazione e Bioingegneria Politecnico di Milano, Italy
{first_name.last_name}@polimi.it

***Abstract*—Image processing applications are today increasingly employed in safety- and mission-critical fields for perception tasks. It is therefore vital to analyse the reliability of the designed system before its deployment and, if necessary, to adopt specific hardening techniques. In this paper we propose a cross-layer reliability evaluation framework specifically meant for image processing applications accelerated onto SRAM-based FPGAs. The framework is based on two key concepts: i) an application-level error simulation based on validated error models to speed-up execution times, and ii) an analysis of the *usability* of the output images based on the working scenario. Such usability analysis allows the designer to study whether the downstream system would be able to take correct decisions even if the image processing outputs are corrupted. We applied the proposed idea on a motion detection application and we compared the achieved accuracy and the required execution times with the ones of a circuit-level fault injector, here considered as a ground truth. This experiment highlighted an accuracy comparable with the one of the fault injection with a dramatic time saving.**

I. Introduction and Related Work

Image processing is increasingly used as the entry stage for processing pipelines enabling autonomous driving and control capabilities in automotive systems, unmanned flying vehicles or robots [1]. Such systems exploit image processing, frequently based on Machine Learning (ML), for perception tasks: images taken from cameras/sensors are processed to extract features that are then used by a downstream control application [2]. Due to their critical role, such applications are required to expose high reliability levels.

Classical redundancy-based techniques, such as Duplication with Comparison or Triple Modular Redundancy, cannot always be applied in a straightforward way on image processing applications. In fact, these applications are highly data- and compute-intensive; therefore, full module replication, applied at any abstraction level (either hardware or software), may not be affordable. On the other hand, image processing is inexact by nature, due to sensor noise and data quantization [3]; therefore, a pipeline is generally designed to tolerate approximation errors. As a consequence, slight modifications of the processed images due to soft errors or other faults may not affect the capability of the downstream control applications to take correct decisions. Indeed, it may be only necessary to take care of those faults that have a disruptive effect on the produced outputs. Based on these considerations, a paradigm shift from the classical bit-wise correct/wrong classification to a *usability-based* one has been proposed in [4], opening the path to novel ad-hoc hardening techniques. In this context, it is fundamental to accurately study the usability of the produced outputs from the point of view of the downstream control application to exploit such new paradigm, leading to a quest for novel advanced in-depth reliability analysis approaches for image processing application.

978-1-6654-1609-2/21/$31.00 ©2021 IEEE

When considering this class of applications implemented onto FPGA devices, the current best practice for reliability analysis consists in the circuit-level fault injection (e.g., [5], [6], [7], [8]). Such approaches generally require the overall system to be fully implemented and deployed on an FPGA, not allowing for a fast and early feedback to the hardening process. Moreover, faults may produce no error or they can be masked, or they may produce an intermediate incorrect data that is later absorbed by the application, thus slowing down the analysis. Finally, most of these approaches [5], [6] are based on the classical bit-wise correct/wrong output classification, while only few ones adopt image quality metrics to measure the visual impact of the fault on the produced image (such as the SSIM index in [7], [8]).

As an alternative to circuit-level fault injection, application-level error simulation is recently receiving interest. In this case, corrupted intermediate data are injected in the application, thus allowing to perform an early and low cost reliability analysis, i.e., no system prototype and target hardware platform are required [9], [10]. Moreover, error simulation ensures that in every experiment the application will be fed with actually corrupted data, thus not incurring in fault activation and masking issues. A possible limitation is related to the accuracy of the results. It is indeed fundamental that the corrupted data injected in the application well represents the effects of the real possible faults, i.e., that the adopted error models are accurate. A first attempt of extracting accurate error models specific for image processing applications accelerated onto SRAM-based FPGAs has been presented in [11].

We present a preliminary proposal of a *usability-based cross-layer reliability evaluation framework* against soft errors for complex image processing applications consisting of a pipeline of filters and implemented onto Xilinx SRAM-based FPGA devices. The framework is based on two key concepts:

- application-level error simulation, and
- usability-based reliability analysis.

The first concept is achieved by means of a two-step process based on i) preliminary fault injection campaigns used to define error model libraries for the most common image

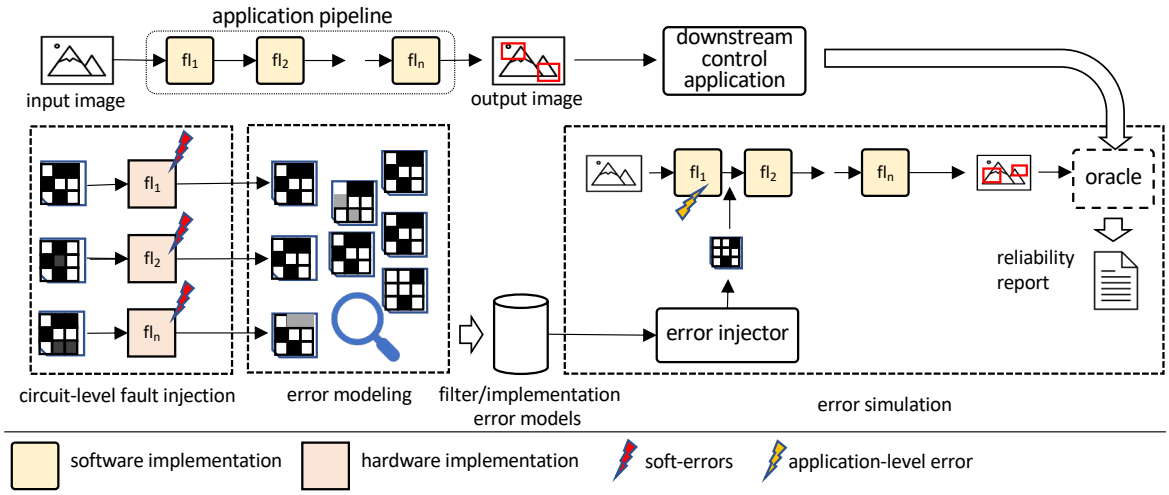

Figure 1. The proposed cross-layer reliability evaluation framework.

processing filters based on the approach presented in [11], and ii) error simulation of the entire image processing application where the previously identified error models are exploited. The second concept requires the definition of a *scenario-based oracle* capable of modeling the main characteristics of the downstream control application and to classify whether the (possibly corrupted) output of the image processing application would still allow the downstream control application to take correct decisions, or not.

The framework has been used to analyze the reliability of a motion detection application, comparing the achieved accuracy and the required execution times with the ones of a circuit-level fault injector, here considered as a ground truth. This experiment highlighted an accuracy comparable with the one of the fault injection experiments with a dramatic time saving.

The remainder of this paper is organized as follows. Section II discusses the details of the proposed framework and its implementation. Section III presents the considered case study and the results from the application of the proposed framework. Finally, Section IV draws the conclusions.

II. The Proposed Framework

Figure 1 depicts a high-level representation of the proposed framework for a usability-based cross-layer reliability evaluation of complex image processing applications accelerated onto Xilinx SRAM-based FPGAs. The framework is designed for image processing applications composed of a pipeline of basic image processing filters, whose output is processed by a downstream control application for decision making. The framework input is the high-level specification of the application (in our prototype we consider a Python implementation). The output of the framework is a detailed reliability report of the performed error simulation campaign offering the possibility to identify the most relevant criticalities of the application w.r.t. the usability of its outputs by the overall system.

The proposed methodology is cross-layer because it combines the accuracy of circuit-level fault injection and the ease and speed of application-level error simulation. The framework is mainly divided into two parts; the leftmost part, that exploits circuit-level fault injection, is devoted to the definition of validated error models relevant for the filters of the pipeline. These models are later used in the rightmost part of the framework to perform error simulation and the usability-based reliability analysis.

Each filter is individually synthesized in hardware on the target platform and analyzed by means of an extensive circuit-level fault injection campaign. The goal is to obtain a rich set of corrupted output images, analyzed in a semi-automated way to extract a set of accurate high-level functional error models representative of all the possible effects of faults in the specific filter/implementation. This activity allows to build a database of error models for the filters/implementations, exploited to perform the reliability analysis of the overall pipeline by means of an error simulation.

The first relevant point is that the error simulation can be performed directly on the high-level software implementation of the application in the early phases of the design flow, thus not requiring the entire system to be synthesized onto the FPGA. Nonetheless, since image processing pipelines are generally composed by a recurrent set of filters, another advantage of the proposed approach is that the error model database is defined only once for each type of filter and implementation on the FPGA platform. In subsequent analyses, it will be possible either to refer to existing models for filter/implementation pairs in the database, or to execute the preliminary error modeling phase. In the long term, all most-commonly adopted filters will be available in the database, and the application analysis activity will only focus on the error simulation part, thus allowing for relevant time-saving.

A second peculiarity of the proposed error simulation environment is how output images are analyzed. Indeed, the classical approach based on a bit-wise check and classification into correct or corrupted result is here replaced by a *scenario-based oracle*. The oracle is in charge of determining whether the global results of the image processing application are

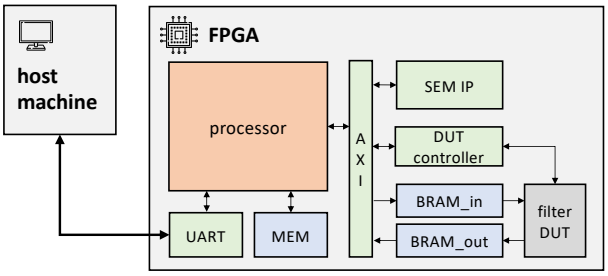

Figure 2. The FPGA circuit-level fault injector.

usable or not based; this classification depends on whether the downstream application using the outputs is able to take the same decision even on partially corrupted data. This approach allows one to focus on faults having a disruptive effect, leveraging on the inexact nature of image processing, where limited corrupted data is inherently tolerated by the processing. The final goal is to reduce as much as possible the overheads introduced by the hardening techniques, possibly having to deal with a limited number of critical cases. The framework integrates an oracle mimicking the downstream control application instead of the application itself for the following reasons: i) the downstream application may not be available when designing and analysing the image processing application, and ii) using the full (possibly complex) downstream application may dramatically slow down the error simulation.

A. Circuit-level Fault Injection

To perform the circuit-level fault injection, the final hardware implementation of each image processing filter is expected to be available, or it needs to be synthesized. In our experimental case study, we used Xilinx Vivado HLS to automatically design the hardware module of each filter starting from its software implementation.

The tool has been designed with a pretty-standard architecture (shown in Figure 2), similar to the one in [8], targeting Xilinx FPGA devices[1]. The hardware implementation of the filter, which represents the Design Under Test (DUT), is integrated in a testbed comprising two memory banks, for the input and output images respectively. A custom module is used to drive the *clock*, *reset* and *start* signals and to receive the DUT *done* output. It is worth mentioning that the testbed needs to be customized based on the specific interface of the current DUT. Finally, the Xilinx SEM IP module is used to emulate soft errors by injecting bit-flips in the configuration memory.

Hardware components are connected to an on-chip processor by means of an AXI bus. The fault injection components are coordinated by a software application running on the processor. It receives from the host machine, through the serial connection, the input image to be used, the golden output and the mask presenting all injectable memory locations and performs a fault injection campaign by following a classical execution flow, where all injectable memory locations are corrupted one at a time, and returns i) the response for each experiment (corrupted output, not corrupted output ,or time-out), and ii) the corrupted output images to the host machine. As a final note, as already discussed in [11], numerous input images are employed during the fault injection campaign not to introduce biases in the identified error models.

[1]The fault injector prototype has been implemented on a Zybo Zynq-7000 development board; however, it can be easily ported on any Series-7 device.

B. Error Modeling

The set of corrupted output images obtained from the fault injection campaign on a specific filter/implementation are then analyzed to define the error models (as proposed in [11]). In particular, the images are visually inspected to identify recurrent visual patterns in the corruption w.r.t. the golden counterpart. Identified visual patterns are also studied to assess that are independent from the specific input sample and can be "reproduced" by an algorithm applied on the golden image. If all these conditions hold, the visual pattern leads to the definition of an *image error model*.

The proposed approach is semi-automated; a tool, implemented as a Python script, compares each corrupted image against the golden counterpart to count the number of different pixels and highlight the areas where such pixels are located. Then, an automated preliminary clustering of the images is performed based on this extracted information; finally, the actual identification of the visual patterns of the errors has been manually performed. The preliminary clustering represents a simplistic hint to the visual pattern identification; then, the final pattern identification requires a relevant effort by the designer.

Finally, the designer has to define an algorithm receiving the golden output as input and capable of mimicking the effects of the faults. Such an algorithmic description may present a parametric fashion since the intensity and the distribution of the corruption frequently vary among the images that have been categorized together for the same visual pattern. The outcome of this phase is a collection of image error models identified for each analyzed filter/implementation to be included in the database to support the error simulation.

C. Error Simulation

Thanks to the accurate error models defined in the previous phases, the overall image processing application can be analyzed against faults by means of an error simulation approach.

The error simulator has been designed in Python, in accordance with the implementation of the image processing application. Figure 3 sketches a simplified version of the main Python script, which contains a high-level description of the error simulator workflow. The loop in the `main` function performs the classical iteration on the number of experiments planned in the error simulation campaign. The loop body executes three main steps: 1) loading of the inputs and golden outputs, 2) execution of the application and error injection, and 3) execution of the oracle to classify the experiment.

As shown in the listing, the application source code is required to be integrated into the main loop; Lines 8-23 present

```
#main loop of the error simulator
def main():
  log = []
  for e in range(0, NUM_EXPS):
    #load data
    [in_img, back_img, golden_out] = load_data()
    #application pipeline + error injection
    gray_img = rgb2gray(in_img)
    [gray_img, err] = inject_error(gray_img, rgb2gray)
    if err is not None:
      corrupted = [gray_img, err]
    gauss_img = gaussian(gray_img)
    [gauss_img, err] = inject_error(gauss_img, gaussian)
    if err is not None:
      corrupted = [gray_img, err]
    mov_img = motion(gauss_img, back_img)
    [mov_img, err] = inject_error(mov_img, motion)
    if err is not None:
      corrupted = [gray_img, err]
    er_img = erosion(mov_img)
    [er_img, err] = inject_error(er_img, erosion)
    if err is not None:
      corrupted = [gray_img, err]
    #evaluate usability
    corrupted.append(usable)
    usable = oracle(er_img, golden_out)
    corrupted.append(usable)
    log.append(corrupted)
  return log

#error injection function
def inject_error(img, filter_func):
  error = selectRndError(filter_func)
  corr_img = applyError(img, error)
  return corr_img

#example of error model corrupting few pixels
def gauss_few_pixels_error_model(img):
  import numpy as np
  rnd = np.random
  noise_num = rnd.randint(1, FEW_PIXELS_NUM)
  out = np.copy(img)
  for i in range(noise_num):
    row = rnd.randint(0, out.shape[0])
    col = rnd.randint(0, out.shape[1])
    val = rnd.randint(1, 256)
    out[row][col] = (out[row][col] + val) % 256
  return out
```

Figure 3. The application-level error simulator.

the image processing pipeline used as case study in this paper, where each filter is implemented by a separate function. As an alternative, the application can be also be executed from an external Python script, to increase the flexibility of the tool. Moreover, the application source code is instrumented with the error injection facilities (Lines 9-11 for the first filter in the pipeline). In particular, since error models are defined as corruptions of the golden output of the selected filter, the injection facility is implemented as a saboteur by the `inject_error` function whose call is located immediately after the code implementing the filter.

When the `inject_error` function is called, it randomly selects an error model for the specific filter/implementation based on the occurrence probabilities characterized in the error modeling phase[2]. Thus, the instantiated error model is applied on the filter output, and the corrupted image is fed to the remaining part of the application pipeline. In a single experiment one filter is corrupted and the corresponding

[2]The implementation of the error model database and the selection of the error to be injected are omitted for the sake of simplicity.

(a) Input (b) Output

Figure 4. Example of input and output images for the case study.

corrupted image is saved to be appended at the end to the campaign log.

For the sake of completeness, Lines 38-48 present also the implementation of an error model performing a random corruption of few pixels in the image. Python NumPy library has been used for image manipulation; being such a library widely used for image processing, it allows the implemented error simulator directly to be applicable to many applications.

The last step of each experiment is the execution of the oracle, whose function is expected to be implemented by designer according to the working scenario to classify results as usable or not. The final output of the error simulator is the log reporting for each experiment the injected error model and the final usability response.

The error simulator may be required to be customized to integrate and analyze a new application and the companion oracle or to add new error models. Being a scripting language, Python has been selected since it gives to the framework the necessary flexibility and customizability of the source code at a very low designer's effort. Moreover, the error simulator is really lightweight; it consists of around 1,000 lines of code, including error models, application and oracle.

III. Case Study

We here present the employment of proposed framework in a case study considering a motion detection application.

A. Case Study Application

The considered case study is an application for the detection of moving objects in a scene. The application has been applied on images taken by a camera on an Italian highway. The output of the application is the list of the bounding boxes to be drawn around the identified cars (Figure 4). The application is composed of a pipeline of four filters:

- an *RGB to gray scale* conversion;
- a *Gaussian* filter to remove the noise in the image;
- a *Motion detection* filter comparing the image against a background one (previously defined with a Gaussian Mixture Model) that produces a black/white image containing the blobs associated with the moving objects; and
- an *Erosion* filter to improve the shape of the blobs.

Finally, a *bounding box* detector computes the coordinates of the bounding boxes and draws them on the original image.

Due to the criticality of the downstream control application that detects the moving cars, the oracle has been defined as in [12] to verify that all identified bounding boxes overlap the

Table I
COMPARISON BETWEEN FAULT INJECTION AND ERROR SIMULATION

	Error Model	Fault Injection		Error Simulation	
		Corrupted	Usable	Corrupted	Usable
RGB to gray scale	multi_pixels	16	16	160	128
	horizontal_shift	4	0	40	0
	horizontal_line	11	3	110	10
	square_shift	83	0	830	0
	few_pixels	428	411	4280	4103
	vertical_shift	21	0	210	0
	vertical_noise	196	58	1960	486
	Total	759	488	7590	4727
	Avg. Usability		64.30%		62.28%
Gaussian	all_black	158	0	1580	0
	border	24	0	240	0
	vertical_line	33	1	330	0
	horizontal_noise	73	6	730	32
	all_gray	8	0	80	0
	vertical_region	18	0	180	0
	colors_corrupted	19	0	190	5
	multi_pixels	19	12	190	189
	vertical_noise_thresh	12	1	120	18
	horizontal_shift	10	0	100	0
	horizontal_line	28	6	280	48
	few_pixels	94	77	940	837
	all_white	4	0	40	0
	horizontal_region	66	0	660	0
	vertical_shift	7	0	70	0
	vertical_noise	50	11	500	24
	Total	623	114	6230	1153
	Avg. Usability		18.30%		18.51%
Motion detection	all_black	4	0	40	0
	multi_pixels	187	145	1870	1855
	thresh_noise	131	6	1310	0
	few_pixels	217	146	2170	1791
	vertical_noise	61	0	610	0
	Total	600	297	6000	3646
	Avg. Usability		49.50%		60.77%
Erosion	all_black	77	0	770	0
	white_border	9	0	90	0
	bb_modify	80	17	800	173
	bb_remove	3	0	30	13
	bb_white_keep	29	0	290	0
	bb_noise	20	2	200	45
	all_white	59	0	590	0
	vertical_noise	19	5	190	26
	bb_shift	16	1	160	0
	Total	312	25	3120	257
	Avg. Usability		8.01%		8.23%

one in the golden output by a Jaccard index measure larger than 50%.

B. Experimental Results

We aimed at validating the accuracy of the proposed error simulation framework and at assessing the advantage of its adoption w.r.t. the classical fault injection process. To this end, we ran a set of fault injection experiments[3] to be considered as a ground truth, and we compared the outputs of these experiments with the ones of a similar set of error simulations. More precisely, we ran 10,000 random fault injections into each filter composing the considered application. Whenever the fault caused the filter output to be corrupted at least in one pixel, we saved the output image. All the collected corrupted images have then been fed in the subsequent filters to analyze image usability. Columns three and four of Table I report the total number of obtained corrupted images after fault injection and the number of usable corrupted images, respectively. We grouped the images w.r.t. the filter under analysis and w.r.t. the error model identified in the corrupted image (as in [11]). The identified error models for each filter are reported in the first column of Table I; it can be observed that some error models, e.g., vertical_noise, are common to all filters, while other error models are specific for one filter or a subset of filters, e.g., horizontal_shift. A first consideration based on these results is that, as it has already been discussed in the motivations for the adoption of error simulation, a very small number of corrupted images has been collected for each filter after 10,000 fault injections (about 3% up to about 7%). To have a significant experimental comparison, we ran (for each filter) a number of error simulations equal to ten times the number of corrupted images obtained by fault injection, by considering the same occurrence distributions of each error model (as reported in the fifth column of Table I). Again, after each error injection we completed the simulation and we asked the oracle to classify whether the obtained output image was usable or not (results reported in the last column of Table I). It is worth noting that the results on usability in the two approaches are very similar. The only exception is represented by the Motion Detection filter, the most complex one, that requires better refined error models to be as accurate. Finally, to show the fidelity of the defined error models, Figure 5 reports some examples of corrupted images for the filters in the application after fault injection and error simulation, respectively. As previously mentioned, faults affecting the Motion Detection filter are the most complex ones to be reproduced; indeed, the defined error models are still not very accurate (especially visible in the last example).

To have an idea of the benefits of the adoption of the proposed error simulation framework in terms of required analysis time w.r.t. fault injection we can notice that, for each of the four filters in the application we injected 10,000 faults; only about 2,200 out of the total 40,000 experiments actually produced a corrupted output, taking around 20 hours. On the other hand, the execution of 22,000 error simulations, all with an error injection, required about 73 minutes. This benefit becomes even more relevant when considering the application of to a real experimental campaign where the number of corrupted images required to build a significant dataset may be hundreds of thousands (e.g., [4]).

IV. CONCLUSIONS

We presented a fast reliability evaluation framework for image processing applications based on accurate error modeling and simulation. The proposed framework allows the designer to perform an in-depth usability-based analysis of the faults that may occur while running the considered application; such analysis, in turn, enables a fine tuning of the fault-hardening technique to be applied, thus increasing efficiency without compromising effectiveness. We applied the proposed idea to

[3]Filters have been synthesized in hardware as in [11].

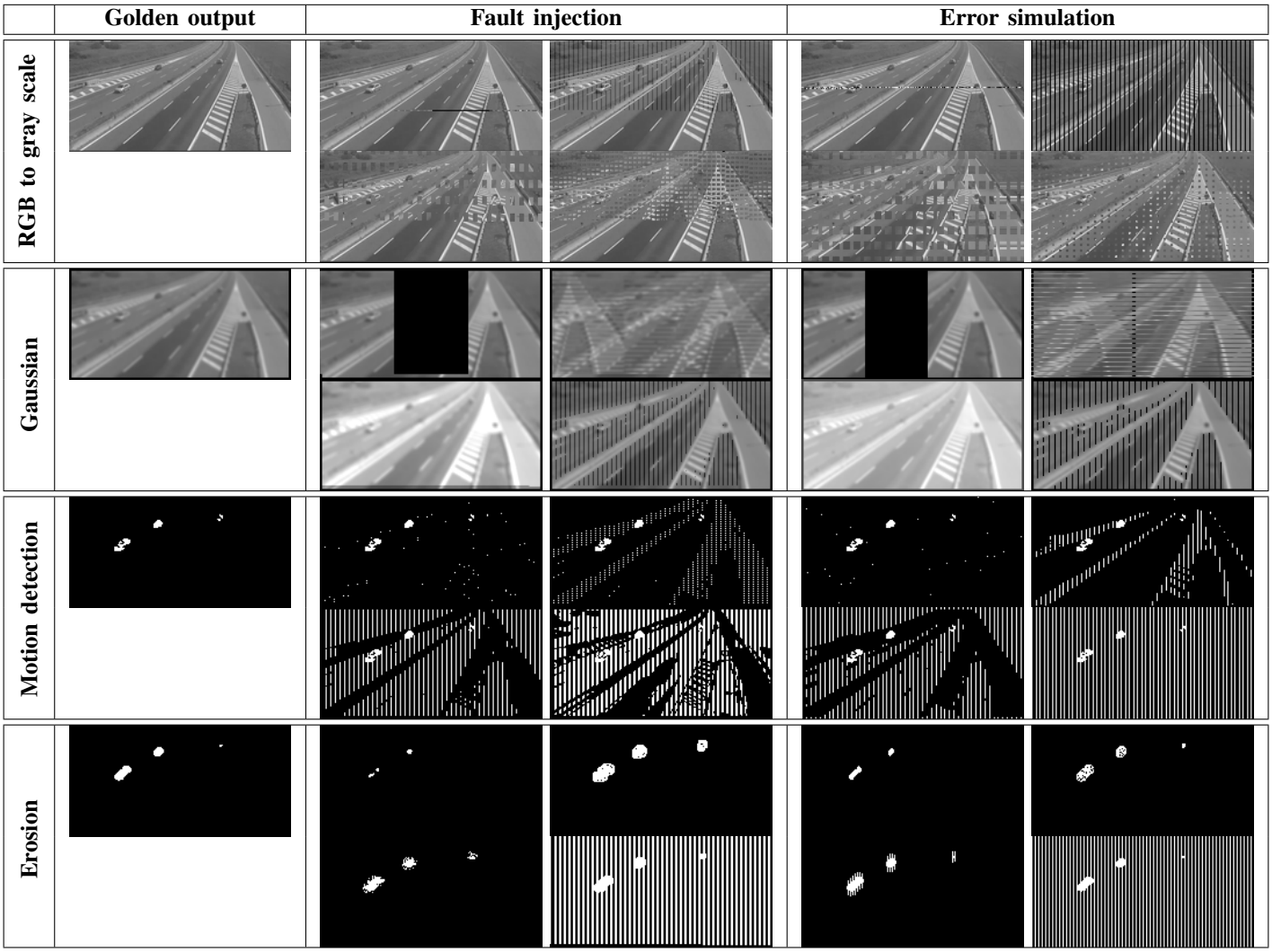

Figure 5. Example of image corruptions generated with fault injection and error simulation.

a motion detection application and we compared the achieved accuracy and the required execution times with the ones of a circuit-level fault injector, here considered as a ground truth. The experiment highlighted a comparable accuracy in terms of the simulated error effects, while requiring a limited amount of time.

REFERENCES

[1] I. Yaqoob, L. U. Khan, S. M. A. Kazmi, M. Imran, N. Guizani, and C. S. Hong, "Autonomous driving cars in smart cities: Recent advances, requirements, and challenges," *IEEE Network*, vol. 34, no. 1, pp. 174–181, 2020.

[2] M. Xu, C. Li, S. Zhang, and P. L. Callet, "State-of-the-Art in 360° Video/Image Processing: Perception, Assessment and Compression," *IEEE Journal of Selected Topics in Signal Processing*, vol. 14, no. 1, pp. 5–26, 2020.

[3] S. Mittal, "A Survey of Techniques for Approximate Computing," *ACM Computing Surv.*, vol. 48, no. 4, pp. 62:1–62:33, 2016.

[4] M. Biasielli, C. Bolchini, L. Cassano, E. Koyuncu, and A. Miele, "A Neural Network Based Fault Management Scheme for Reliable Image Processing," *IEEE Trans. on Comp.*, vol. 69, no. 5, pp. 764–776, 2020.

[5] M. Mousavi, H. R. Pourshaghaghi, M. Tahghighi, R. Jordans, and H. Corporaal, "A Generic Methodology to Compute Design Sensitivity to SEU in SRAM-Based FPGA," in *Proc. Euromicro Conf. on Digital System Design (DSD)*, 2018, pp. 221–228.

[6] B. Du, S. Azimi, C. Sio, L. Bozzoli, and L. Sterpone, "On the Reliability of Convolutional Neural Network Implementation on SRAM-based FPGA," in *Proc. Intl. Symp. on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2019, pp. 1–6.

[7] I. Tsounis, A. Tsigkanos, V. Vlagkoulis, M. Psarakis, N. Kranitis, and A. Paschalis, "Analyzing the Resilience to SEUs of an Image Data Compression Core in a COTS SRAM FPGA," in *Proc. NASA/ESA Conf. on Adaptive Hardware and Systems (AHS)*, 2019, pp. 17–24.

[8] S. T. Fleming and D. Thomas, "Injecting FPGA Configuration Faults in Parallel," in *Proc. Intl. Conf. on Field-Programmable Technology (FPT)*, 2018, pp. 198–205.

[9] Z. Chen, N. Narayanan, B. Fang, G. Li, K. Pattabiraman, and N. DeBardeleben, "TensorFI: A Flexible Fault Injection Framework for TensorFlow Applications," in *Proc. Intl. Symp. Software Reliability Engineering*, 2020, pp. 426–435.

[10] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications," in *Proc. Intl. Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, 2017, pp. 8:1–8:12.

[11] C. Bolchini, L. Cassano, A. Mazzeo, and A. Miele, "Error Modeling for Image Processing Filters accelerated onto SRAM-based FPGAs," in *Proc. Intl. Symp. On-Line Testing Robust System Design*, 2020, pp. 1–6.

[12] F. Fernandes dos Santos, L. Carro, and P. Rech, "Kernel and layer vulnerability factor to evaluate object detection reliability in GPUs," *IET Computers Digital Techniques*, vol. 13, no. 3, pp. 178–186, 2019.