This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2965518, IEEE Transactions on Computers

A Neural Network Based Fault Management Scheme for Reliable Image Processing

Matteo Biasielli, Cristiana Bolchini, *Senior Member, IEEE,* Luca Cassano, *Member, IEEE,* Erdem Koyuncu, *Member, IEEE,* Antonio Miele, *Member, IEEE*

Abstract—Traditional reliability approaches introduce relevant costs to achieve unconditional correctness during data processing. However, many application environments are inherently tolerant to a certain degree of inexactness or inaccuracy. In this paper, we focus on the practical scenario of image processing in space, a domain where faults are a threat, while the applications are inherently tolerant to a certain degree of errors. We first introduce the concept of *usability* of the processed image to relax the traditional requirement of unconditional correctness, and to limit the computational overheads related to reliability. We then introduce our new flexible and lightweight fault management methodology for inaccurate application environments. A key novelty of our scheme is the utilization of neural networks to reduce the costs associated with the occurrence and the detection of faults. Experiments on two aerospace image processing case studies show overall time savings of 14.89% and 34.72% for the two applications, respectively, as compared with the baseline classical Duplication with Comparison scheme.

Index Terms—Dependability, Fault Detection, Image Processing, Machine Learning, Neural Networks.

1 Introduction

There are several classes of systems (e.g., automotive and aerospace) where safety-/mission-critical applications and non-critical applications coexist. For example, in the aerospace domain, the navigation system of a satellite is a mission-critical application while the payload processing applications are not [1]. On one hand, for mission-critical applications, it is mandatory to provide the desired level of reliability in the computation. Thus, fault management mechanisms are generally adopted. To this aim, hardware/-software Duplication with Comparison (DWC) and Triple Modular Redundancy (TMR) are utilized to achieve fault detection and fault tolerance, respectively [2]. On the other hand, since the overall system works in a harsh environment, also non safety-/mission-critical applications may require fault detection/tolerance capabilities, possibly as a *soft requirement*, because errors in the output do not compromise the safety of the system.

Payload applications frequently consist of image processing applications [3]; in the satellite scenario, they collect pictures from a camera, apply a pipeline of manipulating filters and transmit the result to a base station on Earth. These applications may have an inherent degree of fault/noise tolerance, because i) they may deal with noisy inputs (e.g., sensors), ii) their outputs may be probabilistic estimates (e.g. as in machine learning algorithms), and iii) their output images may be used by a human, whose perceptual limitations provide resiliency to a certain level of inexactness [4]. When considering such applications, commonly adopted classical fault detection/tolerance methods (e.g., the DWC) may produce worst-case conservative results since they discard processed data as soon as a single pixel differs. Indeed, in certain cases, the fault that affects the pipeline processing may cause the output image to be only *slightly* altered. The slightly-altered image may still be usable by the end user/application for a correct execution of subsequent processes. In such a scenario, it would be possible to continue the processing and transmission of the result using the slightly-altered image without any re-computation, thus saving time/power [5]. Conversely, if the affected output image is *heavily* altered such that it is not usable by the end user, it is crucial to be able to detect the fault and re-execute the processing as soon as possible to limit time/power overheads. Therefore, in such a context, deciding whether to discard or not an image based on the occurrence of a fault is not enough. At the opposite, it would be beneficial to be able to decide if the overall image is still usable by the end user/application to decide whether to discard it or not.

In this paper, we propose a time-efficient flexible fault management scheme that extends the classical DWC approach by exploiting the classification capabilities of Convolutional Neural Networks (CNNs) to discriminate between usable and unusable application's outputs. When a fault causes an error, the effects are evaluated and only if deemed necessary, the application is re-executed to mitigate the effects of the fault. Our new method aims at limiting the reliability-related overheads by avoiding unnecessary re-executions, thus saving time and power. While the proposed scheme is general, the core component realizing the flexible error detection strategy needs to be designed, trained and optimized specifically for the application under consideration. To this end, a design methodology is also presented, together with a semi-automatic framework supporting the

- *M. Biaselli, C. Bolchini, L. Cassano and A. Miele are with the Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy (E-mail: {matteo.biaselli, cristiana.bolchini, luca.cassano, antonio.miele}@polimi.it).*
E. Koyuncu is with the Department of Electrical and Computer Engineering, University of Illinois at Chicago, Chicago, IL, USA (E-mail: ekoyuncu@uic.edu).
Authors appear in alphabetical order.

Manuscript received April 19, 2005; revised August 26, 2015.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2965518, IEEE Transactions on Computers

designer in the implementation of the hardened image processing application. A preliminary version of the approach has been presented in [5]; we here extend it to achieve a mature proposal that includes these contributions:

- an improved tunable Smart Checker (SC), based on a refined CNN and a *Difference Evaluator* to invoke it; the last component replaces the Two-Rail Checker (TRC) module of the previous proposal;
- a systematic approach to instantiate and tune the parameters of such a checker;
- a semi-automated design flow to select the CNN internal architecture, train it and optimize its execution time by pruning the irrelevant parts in its internal structure;
- an extensive experimental campaign to demonstrate the effectiveness of both the proposed fault management scheme and companion design flow.

We applied the proposed flexible fault management scheme to two realistic aerospace image processing applications. Results show an overall time saving of about 14.89% and 34.72% for the two applications, respectively, incurring a reliability-related penalty (not usable images that have not been discarded) of about 0.3% and 1.6%.

The rest of this paper is organized as follows: Section 2 discusses the related work, and Section 3 introduces the motivations behind the design of this scheme. Then, Section 4 discusses in details the proposed fault detection scheme for image processing and presents the methodology for hardening a target application by means of the proposed scheme. In Section 5, we first introduce the two case studies to analyze and validate the proposed solution and then we discuss the obtained experimental results. Finally, Section 6 concludes the paper and highlights future work.

2 Related work

The idea of adopting hardening schemes that relax some constraints has already been explored in various directions in the literature and at different levels of abstraction, from logic level to Register Transfer Level (RTL).

Various approximate/inexact TMR schemes have been proposed at the logic level [6], [7], [8]. They mainly aim at reducing the area required by the hardened circuit by trading-off the precision of the detection/mitigation scheme. For example, the classical TMR scheme is applied to a single-output combinatorial function, using two replicas that are smaller but introduce errors, masked by the subsequent majority voting scheme. Different fault detection/management schemes have been considered in [9], starting from the DWC, and other are presented in [10], devoted to reduce the size of the checker/voter circuits. Common to these approaches is their focus on the single bit or value being manipulated, such that these proposals work at a low abstraction level to tailor the implemented function, making it difficult to apply them at the application level with multi-dimensional data.

Other approaches [11], [12] acting at RTL are based on the reduction of the numeric precision in the DWC and TMR schemes. The arithmetic units of the additional replicas and the checking/voting modules elaborate and check only the most significant bits of the values in the nominal circuit so that protection is provided only on a subset of the bits, the most relevant ones for the result of the computation. The rationale is that the application context is inherently tolerant with respect to a certain degree of impreciseness in the results, due in reality to the presence of an error on the least significant bits. Our proposal stems from the same consideration, but the application scenario we take into account does not allow for these techniques to be applied, being all parts of the image relevant in the same way.

An enhanced version of the same strategy consists in replacing the classical checker in fault detection schemes with a comparison of the numerical distance between the outputs of the nominal system and the introduced possibly-approximate replica with respect to a given threshold. This approach proved to be suitable for Floating Point Unit [13] and for Digital Signal Processing circuits [14], [15]. To some extend, these solutions introduce the idea of usability of the slightly corrupted results we here exploit. Indeed, although the authors state that their solution is applicable to the image processing context, there is no straightforward extension from their scenario based to the single scalar data to the image one. It has also been observed that designing a system to provide an acceptable/usable output rather than thest-possible/correct output can yield significant gains in terms of the system's implementation costs and complexity [16], [17]. The solutions are however not applicable to our scenario due to the entirely different settings.

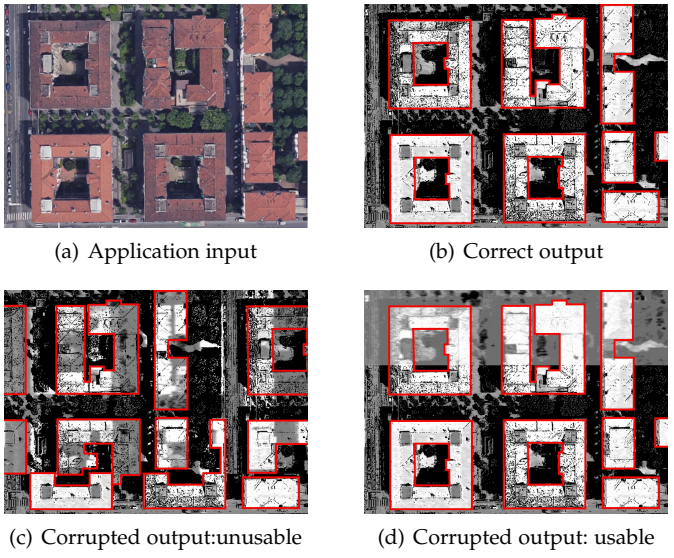
The preliminary work at the basis of this proposal has been presented in [5], exploiting a smart fault management scheme and introducing the usability concept. The approach employs CNNs to have an advanced checker capable of analyzing the effect of the fault on the final result of an image processing application. However, the CNN is simple and is invoked every time there is a single different pixel between two replicas due to the occurrence of a fault, limiting the performance benefits. This new proposal enhances that idea and introduces a flexible, tailored and optimized checker, defines the methodology and the software framework to harden the application, achieving improved results with respect to such previous work.

3 Motivation and Preliminaries

Let us consider the case of an image processing application consisting of several steps, running on a single-core microprocessor in an environment subject to radiation effects. Such radiations may produce Single Event Upsets (SEUs) that affect the process(es) that manipulate the image, causing the overall application output to be corrupted. As an example, consider an application for identifying buildings in images taken from a satellite (Fig. 1(a)), by highlighting them with bounding boxes (Fig. 1(b)). Should a fault occur during the processing and produce an observable error on the output image, one of the two situations will hold:

i) the impact of the fault is relevant, such as the one reported in Fig. 1(c), and the identified bounding boxes differ from the ones extracted in a fault-free condition;
ii) there is a visible effect of the fault, as shown in Fig. 1(d), however the corrupted image is still usable leading to a final result (the bounding boxes) that is basically identical to the fault-free output.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2965518, IEEE Transactions on Computers



(a) Application input (b) Correct output

(c) Corrupted output:unusable (d) Corrupted output: usable

Figure 1. Fault effects on processed images leading to unusable (1(c)) or usable (1(d)) images. Images are taken from Bing maps [18].

Table 1
Performance evaluation: notation and classification

D	Discarded	the SC classifies the data as not usable
/D	Not Discarded	the SC classifies the data as usable
U	Usable	the data is usable
/U	Unusable	the data is not usable
D /U	Correctly handled corrupted data	
/D U	Achieved execution time savings	
D U	Not exploited execution time savings – false positive	
/D /U	Erroneously accepted data – false negative	

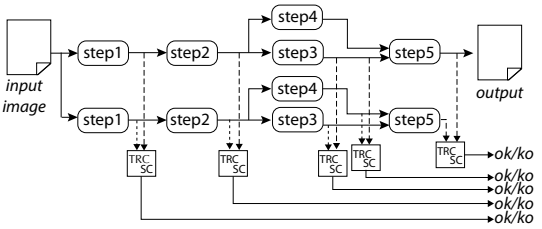


Figure 2. The fault detection scheme: the classical DWC solution is extended by replacing the TRC with an SC module.

The proposed approach discards a corrupted image (intermediate data or final output) and performs re-computation only when the image is deemed not usable by the downstream application or end user.

While the difference between correct and corrupted is well-defined, the difference between usable and unusable is often context-related. As such, our proposed approach has a stochastic nature and may incur in misclassifications: i) a usable image is discarded (false positive), or ii) an unusable image is accepted (false negative). False positives impact the effectiveness of the proposed approach, because re-computation is performed when not needed. On the other hand, false negatives should be avoided because they affect the reliability. However, since the methodology is envisioned for not mission/safety-critical tasks, they lead to no dangerous situation.

3.1 Evaluation criteria

The methodology and its companion design flow take into account various alternatives and, based on their performance, select the most promising option. For intermediate steps and for the final solution, the effectiveness is evaluated in terms of i) the correctness of the classification with respect to the usability of the image, referred to as *performance* in the following, and ii) the execution time saving. Performance is evaluated by feeding the SCs of the final complete system with corrupted images and by computing the percentages of them classified as to be discarded (D) because considered too corrupted, or not discarded (/D). This classification is analyzed against what the images really are, usable (U) or unusable (/U). Table 1 reports the resulting four classes.

3.2 Working scenario

We propose a software-based solution for achieving fault tolerance in image processing applications executed on a general purpose embedded CPU, mitigating the effects of SEUs in the microprocessor memory elements or in the system memory. Faults in microprocessors may lead to i) application crashes or operating system exceptions blocking the execution, ii) application hangs leading to a non-termination, and iii) silent data corruption, where the executed application terminates producing an erroneous output, without any alert [19]. The first class is managed autonomously by the operating system and the second one can be dealt with by means of watchdog mechanisms; the last one represents the most critical situation requiring an ad-hoc hardening, addressed here. We assume a single SEU may occur during a run of the application, also in harsh environments, thus every fault is an independent event.

The proposed methodology uses a mix of space and time redundancy to achieve fault tolerance, by exploiting and extending the DWC scheme to detect the occurrence of a fault, and, when necessary, by performing a re-execution of the computation. The classical DWC uses a TRC to perform a bit-wise comparison, such that any single corrupted pixel leads the image to be re-processed. The re-processing is performed independently of the impact of the corruption on the image, that is the number of corrupted pixels and effect on the usability of the produced output. Our contribution replaces the TRC with a more flexible SC (Fig. 2).

The solution we propose is general and can be applied to other hardware/software platforms, provided the fault/error relations are opportunely adapted, to enable the flexible identification of errors that can be tolerated. Future work will investigate this extension.

4 A Flexible Fault Management Approach

This section introduces the SC architecture and its design methodology. Each SC is meant to handle different intermediate data, therefore it is independently designed and customized for the application step it analyzes. Although it is not possible to design a single generalized module to fit all SCs, the methodology and the practical process to implement them is systematic and the application developer can rely on it to obtain the custom SC modules.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2965518, IEEE Transactions on Computers

4.1 The Smart Checker architecture

Fig. 3 shows the internal structure of the Smart Checker (SC) of each one of the DWC blocks depicted in Fig. 2 (where for readability reasons, not all inputs of the module are shown). The module receives the application input image and two replicas' outputs and determines whether a re-computation is necessary (the output is a Boolean `ok/ko` signal). Given a set of inputs, the following situations may arise:

- the replicas' outputs are identical (no error occurred),
- the two replicas' outputs are *slightly* different (a fault corrupted one of them), or
- the two replicas' outputs are *heavily* different (a fault corrupted one of them) and it is necessary to further investigate whether they are still usable or not.

The *Difference Evaluator* block computes the difference between the two replicas' outputs, value by value, and the percentage of such differences. If the percentage is above a set α threshold the CNN is invoked on the application input image and the difference of the two replicas' outputs, possibly after resizing. As illustrated in Fig. 3, the two resized inputs to the CNN are stacked as different sets of input channels to the CNN: For example, if the colored input image has the three RGB channels, the CNN will have a 6-channel input, the 3 channels of the resized input image and those of the resized difference image. We employ CNNs because they are the state-of-the-art in image feature extraction and classification tasks. Here though, the CNN is exploited in a different way, to determine the usability of the intermediate data and the final output.

The defined two-step analysis is particularly promising since it combines the reduced execution time of the simple *Difference Evaluator* with the more accurate classification of the time-demanding CNN. As a consequence, the selection of a proper α threshold is particularly relevant. Such value is empirically identified, based on the application scenario and the usability of the corrupted images, and is determined at design time, when training the CNN and the SC itself. In particular, α is set to a value such that when the difference is below, the output of the overall process is still usable by the downstream application. When the difference is larger than α, it may be necessary to re-process the image. In the latter case, the CNN will make the final decision. While the solution in [5] is based on a *Difference* module that computes a per-pixel difference and triggers the CNN classification every time there is a difference in the replicas' outputs (i.e., acting as a TRC).

Finally, to limit the complexity of the classification process and to reduce the execution time of the CNN, a resizing block may be added to reduce the input image and the difference of the two replicas. As we shall also demonstrate in Section 5, resizing up to a certain downsampling rate may provide a better overall tradeoff between accuracy and execution time. For applications that involve high-resolution image processing, resizing may degrade the overall performance and thus can be omitted.

As discussed, the CNN is the most expensive component in terms of computational complexity. Therefore, it is invoked only when deemed necessary, because the time savings for not re-executing the processing must be balanced by the time for running the SC. To this end, we have designed a general SC architecture presented in Fig. 3 to be customized based on the stage of the image manipulation process. The customizable items are 1) the threshold α of the *Difference Evaluator*, and 2) the CNN.

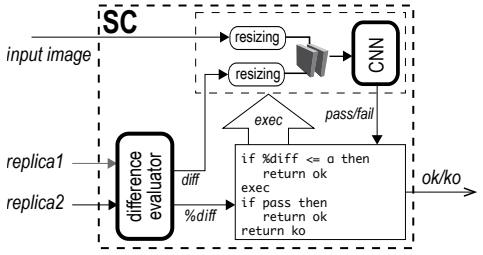


Figure 3. The proposed Smart Checker module.

This architecture includes both the traditional DWC scheme (corresponding to setting $\alpha = 0\%$ and having no CNN block), and the solution presented in [5] (corresponding to setting $\alpha = 0\%$ and having a CNN block, i.e., invoking the CNN every time at least one pixel differs). Should the cost of invoking the CNN be considerable with respect to re-executing the application step, it is possible to use the *Difference Evaluator* and trigger a re-execution anytime the difference between the replicas is above a given α (corresponding to $\alpha > 0\%$ and no CNN block). This is a limit case for the proposed methodology, therefore we will focus on the more general case.

The identification of the most effective α value in relation to the CNN, and the CNN design for each possible processing step are discussed in the following.

4.2 Design Methodology

The key elements of the SC are the *Difference Evaluator* and the CNN. The specific choice of the threshold α affects the operation of both modules and thus the overall system performance. In fact, a high α would reduce the number of times the CNN is executed, thus increasing the time savings. Indeed, such a choice would also potentially increase the number of erroneous decisions taken by the SC. This is because a high α would increase the number of images that are accepted/discarded by only counting the number of different pixels, without resorting to the "intelligence" provided by the CNN. On the other hand, a low α would represent a too conservative choice: In this case, the CNN would be executed also when the two images are almost identical, thus reducing both the number of errors but also the time savings. The CNN needs to provide high accuracy in classifying usable/unusable data. On the other hand, a shorter CNN execution time is also desirable for time savings. In fact, to reap any time saving benefits through CNNs, a necessary condition is that the time saved by reducing the number of unnecessary re-executions is more than the time overhead incurred by the execution of the CNN itself.

We adopt the design flow depicted in Fig. 4 to design and optimize the α threshold value of the *Difference Evaluator* and train the CNN. The process is executed once per each step of the application to specifically tailor the SC module. Inputs to the flow are i) the application to be analyzed, ii) the hardware architecture where the application is executed,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2965518, IEEE Transactions on Computers

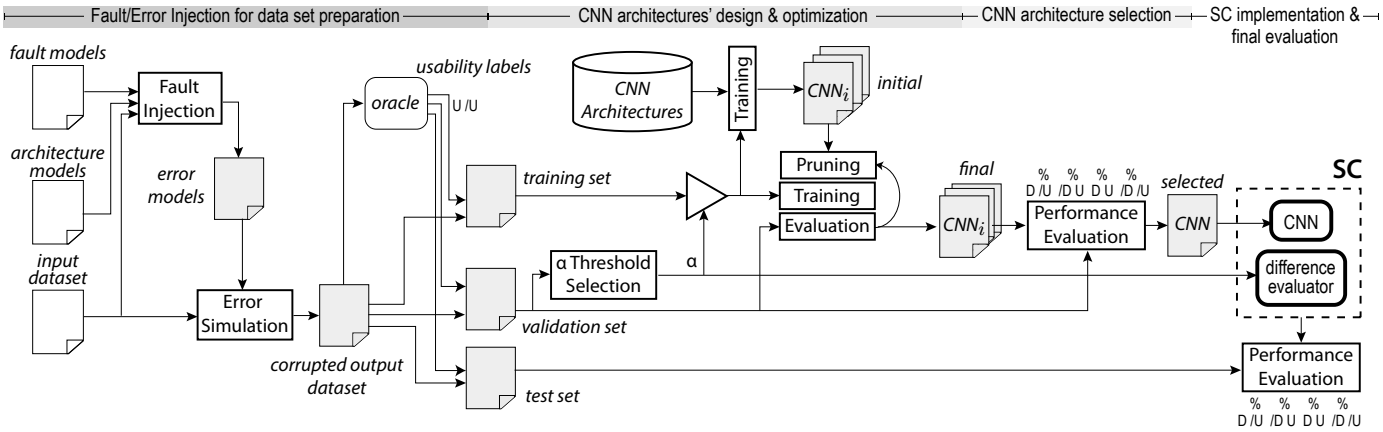


Figure 4. The SC design methodology and flow.

iii) the fault models appropriate for the HW platform, iv) a set of input images representative of the images the application is meant to process, and v) an *oracle* that mimics the final user of the results of the application under analysis. The oracle is a tool that implements the post-conditions of the considered image processing application, that are the requisites for the usability of the inputs for the downstream end-user application.[1] The first stages of the design flow (namely fault injection and error simulation) are used to generate the training, validation and the test sets, used to design and evaluate the SC. The best α threshold value is chosen and several CNN architectures are trained and then evaluated by means of the validation set. Once the best CNN is identified, it is pruned to reduce its execution time, and its performance is evaluated using the test set.

The design space considering all parameters at play is too wide to be explored exhaustively, therefore the adopted flow selects, in each phase, the local optimum. This strategy does not guarantee the optimality of the overall solution, however experimental results show that, in general, the final design performs better than the discarded alternatives.

4.2.1 α Threshold Selection

The first stage of the design flow is to determine the best α threshold value for the *Difference Evaluator*. First, a set of candidate α values is identified. Then, for each α value and for the specific training set, all images are processed by the *Difference Evaluator* so configured, and we collect the information of the class they belong to, among /D U, D /U, D U and /D /U. The ultimate goal of the flexible fault management scheme is to benefit from situations where a usable image is not discarded, thus α could be selected so that /D U is the highest among all alternatives. However, the higher the threshold, the higher the probability to erroneously classify the image as usable when it is not. In this work, we want to avoid false negatives, therefore the selected α is the one that achieves the highest portion of /D U with the lowest portion of /D /U, possibly 0%. Hence, there is one and only one choice of α for the application requirements considered in this work. In a relaxed application context, it could be more convenient to accept a limited amount of false negatives to achieve higher performance savings.

1. The oracle can be replaced by the end-user application if available.

All pixels are treated uniformly; it could be possible to use other policies should the application and/or the images allow for it. The *Difference Evaluator* would be further specialized to trigger the re-computation in a more tailored way, an opportunity we will investigate in the future.

4.2.2 CNN Design and Optimization

This design stage explores several CNN architectures to determine which one is the most suitable to work in conjunction with the chosen *Difference Evaluator*. We follow the standard and widely adopted guidelines for CNN architecture definition [20] still keeping in mind that there is no "golden" network architecture that is guaranteed to work in the best way for a specific task.

We adopt a CNN architecture composed of a number of convolutional layers interleaved with max-pooling layers, which perform dimensionality reduction and help pushing the model towards better generalization. Also, the last layer of the CNN is a fully-connected layer that performs the final classification [21], [22], [23]. To obtain binary classification, the activation function for the last layer is always the Sigmoid function. All the other layers use the Rectified Linear Unit (ReLU) activation function and the dropout rates are always chosen between 0.25 to 0.5, to prevent overfitting [24]. The CNN architectural parameters that have to be tuned are the number of convolutional layers and the number of filters per convolutional layer.

4.2.3 CNN Training

We train the defined CNNs with the Adam optimizer [25], that exploits the benefits of both AdaGrad [26] and RMSProp [27] strategies. The considered classes consist of images that are deemed by the end user either usable and acceptable (U) or unusable and rejected (/U). As our CNNs deal with a binary classification problem, the objective loss function used by the optimizer can be chosen to be the *binary cross-entropy*

$$\sum_{s \in S} -(w_U y_s \log(p_s) + w_{/U}(1 - y_s) \log(1 - p_s)) \qquad (1)$$

with class weights $w_U, w_{/U}$. Here, S is the sample set, y_s is the label of sample s, and can only assume 0 (reject) or 1 (accept). Also, p_s is the output of the CNN, representing the likelihood that an image should be accepted.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2965518, IEEE Transactions on Computers

The weights w_U and $w_{/U}$ represent the impact of misclassifications of samples of the corresponding class on the loss function. When $w_{/U} = w_U$, every misclassification has the same impact on the loss function. When $w_{/U} > w_U$, the training will be biased towards reducing the cases where a /U image is classified as U. In other words, setting $w_{/U} > w_U$ attempts to minimize the number of /D /U cases, although potentially increasing the number of D U.

In our design flow, we trained each CNN with the latter configuration ($w_{/U} > w_U$) and optimized the ratio $w_{/U}/w_U$ to minimize the amount of /D /U while still keeping the D U percentage low. To prevent overfitting, *Early Stopping* [28], *Dropout* [24] and *Ridge Regularization* [29] are employed.

The CNN is trained only on the set of training images that are rejected by the difference evaluator. A similar design choice is considered in the context of multi-stage conditional neural networks [30], [31]. In particular, in [30], a given stage of a multi-stage CNN is trained using only the training samples that are passed from the previous stage. Still, another option may be to use the entire training set for training the CNN. In this case, the resulting CNN should also learn to correctly label the images that are accepted by the difference evaluator as "usable," increasing the overall complexity of the SC. Another disadvantage of the latter option is increased training time due to a larger set of training samples. Intuitively, training a CNN over a set of images that it will never encounter translates to a waste of neural computing resources.

4.2.4 CNN Pruning

Pruning a CNN means modifying it by removing connections, neurons or convolutions. This reduces the number of parameters and, as a consequence, the execution time. The expected execution time reduction typically ranges from 3x to 11x. We applied the following two algorithms [32]:

- *Average Percentage of Zeros* (APoZ): the connections in a layer that most often produce zeros are pruned.
- *Sum of Absolute Weights* (SoAW): typically, very few connections in a CNN have large weights, while many connections have small weights. The former type of connections are the ones that affect the final classification the most. Under this assumption, it is possible to prune the low-weight connections without impacting the classification accuracy.

After pruning, the CNN needs to be re-trained to allow the remaining connections to adjust their weights.

We designed an automated and iterative CNN pruning process that applies the APoZ and the SoAW algorithms, retrains the pruned CNN and evaluates the new classification accuracy by using the validation set. In case the difference between the original accuracy and the new accuracy is lower than a given threshold, the pruned CNN is accepted and a new pruning iteration is attempted. If the difference in accuracy exceeds the threshold the pruning procedure ends.

Regarding the complexity of the pruned CNNs, we note that our applications involve several non-trivial steps in general. SCs operate at each step to decide whether the final output of the system, after passing through all remaining steps, would be usable or not. Therefore, CNNs of the corresponding SCs should necessarily be "complicated enough" to capture the effects of the non-linearities that will follow.

Algorithm 1 The training set generation procedure

Inputs:
DS : image dataset
s_i : application's step to be analyzed
Outputs:
TS : computed training set

Body:

1: $TS \leftarrow \{\}$
2: **for all** $I \in DS$ **do**
3: **for** $e \leftarrow 1$ **to** E **do**
4: $r_i \leftarrow$ run_application_steps(I, s_1, s_i)
5: $r_i^* \leftarrow$ inject_random_error(r_i, s_i)
6: $r_N^* \leftarrow$ run_application_steps(r_i^*, s_{i+1}, s_N)
7: $r_N \leftarrow$ run_application_steps(I, s_1, s_N)
8: $label \leftarrow$ oracle(r_N^*, r_N)
9: $TS \leftarrow TS \cup \langle I, r_i^*, r_i, label \rangle$
10: **end for**
11: **end for**

4.2.5 Fault Injection and Error Simulation

This stage of the design flow generates the training, validation and test sets for the CNN starting from a set of input images that are representative of what the application will process at runtime. We split the fault injection experiment from the error simulation one to be effective in terms of the overall time required to carry out the training, validation and test sets generation. Indeed, when performing fault injection on a microprocessor, it might occur that the fault either causes a crash or timeout of the application (we do not consider such cases, as previously discussed) or it does not affect the output. Thus, the number of corrupted outputs may be small with respect to the number of fault injection experiments. On the other hand, for an effective CNN training, a large number of corrupted output data is required.

Given a specific application step, we first perform a fault injection campaign to collect a significant number of corrupted intermediate outputs. This phase is performed by using a fault injection tool capable at simulating both the considered architecture and fault models (which are the target microprocessor architecture and the SEUs, respectively, in the considered scenario). Each corrupted output consisting in a distortion of the nominal result is representative of the different ways a fault can affect the computations of the considered processing step. Based on these corrupted images, we manually extract distortion patterns and we categorize them into a set of representative error models that we use in the subsequent automated error simulation campaign. Differently from fault injection, in error simulation, every experiment produces a corrupted result.

Error simulation is then employed in the procedure reported in Algorithm 1 to generate the training set for each step s_i of the considered application. For each image in the input dataset, the algorithm simulates E different random errors to produce a corresponding number of items to be included in the training set TS (Lines 2–3). The input image

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2965518, IEEE Transactions on Computers

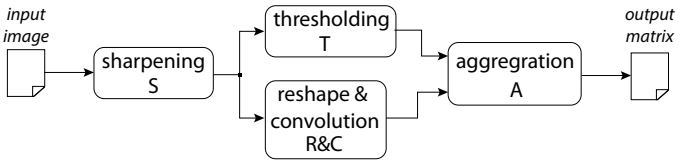


Figure 5. The structure of the Buildings Identification Application.

I is first processed by using the previous application's steps from s_1 to s_i, an error is randomly selected among the available models for the current step s_i and injected in the intermediate result r_i (Lines 4–5). The corrupted intermediate output r_i^* is then elaborated by means of the remaining steps from s_{i+1} to s_N to obtain the final corrupted image r_N^* (Line 6). A golden final output r_N is also computed by running the entire application, so that the pair r_N^* and r_N are used as input for the oracle to classify the experiment with a usable/unusable $label$, i.e. U or /U (Lines 7–8). Finally, the obtained tuple $\langle r_i, r_i^*, I, label \rangle$, where the three items represent an instance of possible inputs of the SC and the $label$ the desired classification, is added to the training set TS (Line 9). The test set and the validation set are generated by following the same algorithm.

5 Experimental Results and Discussion

In this section we first describe the experimental setup, then we report the results for the two application case studies.

5.1 Experimental Setup

The design process of the SCs is realized by a Python framework that implements for each SC, its modules, namely the *Difference Evaluator* (dubbed *DiffEv* in the tables) and the CNN. For CNN design and training, TensorFlow and Keras Python packages have been adopted [33].

The fault injection campaign is carried out using LLFI [19], which can inject hardware faults mimicking SEUs in the application software being executed, targeting the processor and the memory (e.g., instruction's operand bit-flip, data corruption). We ran a campaign for each application's step, identifying for each one a set of error models that can occur on the output. As a result, the error models are associated with the specific application steps being executed and the hardware architecture on which it is run.

The applications are implemented in C++, as well as the final deployed SCs, and all performance evaluations (i.e., execution times) are collected on an Odroid Board XU3 with a single-core Arm A15 architecture and tasks scheduled in a time-triggered fashion.

For each case study, we discuss the process for the identification of the α parameters of *Difference Evaluators*, and the CNN design and pruning phases. Finally, we report the execution times and performance gains of the fault management scheme, in comparison with respect to the classical DWC solution and the one presented in [5].

5.2 Case Study #1: Buildings Identification

Our first application identifies buildings in aerial images. The application accepts a bitmap and produces a heatmap of the same size where each input pixel in the input image is encoded with the probability of the pixel to belong to a building. Bounding-boxes are then drawn on the heatmap to highlight the identified buildings. An example input image and the corresponding output are shown in Fig. 1. The block diagram of the application is illustrated in Fig. 5. The application is composed of four steps: i) *sharpening* (S) is an convolution that applies a sharpening kernel. This step takes an image and produces the corresponding sharpened image; ii) *thresholding* (T) performs a pixel-wise classification by doing a channel-wise comparison of the values; iii) *reshape&convolution* (R&C) performs a classification based on image reshaping and then a convolution. The two precious classification steps take the sharpened image and produce two matrix of probability values. iv) *aggregation* (A) performs a pixel-wise multiplication between the outputs of the two previous steps to aggregate their classifications. This step takes two matrices and produces the output heatmap.

As a simple end-user application for the building identification, we implemented an oracle that given a (possibly faulty) heatmap and the corresponding correct one outputs a *usable* label if the bounding-boxes on the two heatmaps overlap for at least the 85% of the total area.

5.2.1 Training/Validation/Test Set Generation

For our first case study, we generated a dataset of input images by collecting about 1,000 aerial images of a set of cities from Bing maps (by using the available API [18]).

We performed fault injection experiments with LLFI in the four steps of the building identification application. The identified error models, similar for the different stages, are: i) *image shift*, a portion of the output matrix is shifted; ii) *black area*, a portion of the output matrix is set to black; iii) *black spots*, pixels, lines or squares are set to black; iv) *color change*, a constant change in the likelihood index of a part of the output matrix occurs (on an image it appears as a change in the luminosity, hence the name).

We generated a training dataset for each application's step based on the procedure in Algorithm 1; in particular, we corrupted each one of the 1,000 input images by simulating 10 different error models, generating a training set constituted by 10,000 corrupted images. Similarly, we generated the validation and test sets, starting from other 300 and 1,000 images, respectively, from different cities and simulating 10 errors per each one.

5.2.2 α Threshold Selection

Each one of the four *Difference Evaluator* blocks is designed and tailored to allow corrupted images not to be discarded provided the CNN is able to correctly decide whether it is necessary to repeat the processing. The customizable parameter is the α value of the percentage of the number of different pixels between the two replicas, such that no false negatives (/D /U) occur. We have explored integer values of $\alpha \in [0, 10]$, analyzing both the distribution of the classified images in one of the four classes, and the execution times, comparing the results against the baseline (the TRC module, adopted in both the classical DWC technique and the one presented in [5]). All experiments have been executed on a set of 10,000 samples, and for each α value, we analyzed the behavior of the *Difference Evaluator* with respect to its

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2965518, IEEE Transactions on Computers

Table 2
Corrupted image management for different α values for the R&C step

	D /U	/D U	D U	/D /U
$\alpha = 0 \equiv TRC$	54.46	0.00	45.54	0.00
$\alpha = 1$	54.46	34.81	10.73	0.00
$\alpha = 2$	54.46	35.86	9.68	0.00
$\alpha = 3$	54.46	36.72	8.82	0.00
$\alpha = 4$	54.46	37.54	8.00	0.00
$\alpha = 5$	54.46	38.15	7.39	0.00
$\alpha = 6$	54.46	38.77	6.77	0.00
$\alpha = 7$	54.46	39.30	6.24	0.00
$\alpha = 8$	54.45	39.81	5.73	0.01
$\alpha = 9$	54.45	40.46	5.08	0.01
$\alpha = 10$	54.41	40.74	4.80	0.05

Table 3
*Difference Evaluator*s Characteristics – Case Study #1

	S	T	R&C	A
α	1%	2%	7%	2%

decision to keep the image or not, and by observing whether the final output was indeed usable.

Results for the *Difference Evaluator* on the R&C step are reported in Table 2. The first column identifies the adopted α threshold; the subsequent four columns report the percentage of images classified as D /U, /D U, D U (false positives) and /D /U (false negatives), respectively. Following our discussion in Section 4.2.1, we can observe from Table 2 that the best choice is $\alpha = 7\%$ since it provides the highest flexibility (i.e., the highest number of /D U) without incurring false negatives.

The same process has been carried out for all the four steps, collecting similar data; the adopted final implementations of the four *Difference Evaluator*s are reported in Table 3. The execution time for this module is 4.7ms, the same execution time required by the TRC module.

5.2.3 CNN Architecture Design and Training

Once the *Difference Evaluator*s are tuned, the *Resizing Block* and the CNN are designed in a single phase. We explored different feedforward CNN architectures.

Architectural optimization can significantly improve the performance of any neural network. In this context, we note that our work is the first to utilize neural networks for deciding image usability in reliable image processing applications. The input channels to the neural network are the input image and a difference map obtained at a certain application step. While the best neural network architectures for usual image classification tasks (where the input to the neural network is a mere image) are well-understood, finding the best network architecture for our specific scenario is thus an open problem. Our proposed solution here is a simple grid-like search where we try certain well-known network architectures, followed by pruning. One can also potentially utilize some of the existing neural architecture search algorithms [34], [35] for finding an appropriate neural network topology for the SC. Extensions of our results in this direction are left as future work.

Our networks differ in the amount of filters in their convolutional layers, to analyze how the number of extracted features impacts the final accuracy. Among the several explored solutions, we report here the most effective implementations. In all networks, the last two layers are fully connected layers with 25 neurons and a single neuron, respectively.

Table 4
Execution times of different SCs architectures – Case Study #1

	const_filters	*fat_to_thin*	*thin_to_fat*	*bell*
Time (ms)	33.2	98.3	30.9	18.1

- *const_filters*: This architecture has the same number of filters in each convolutional layer. The network configuration is 2+3+2, meaning that the sequence of network layers is given by two convolutional layers, a max-pooling layer, three more convolutional layers, a max-pooling layer, and two more convolutional layers (and the two fully-connected layers as mentioned previously). Each convolutional layer has 20 filters.
- *fat_to_thin*: In this architecture, the number of filters decreases at each subsequent layer of the network. The network configuration is 2+2+2. The convolutional layers have 40, 35, 30, 30, 30, and 20 filters, respectively. The rationale is to extract as many features as possible in the beginning and then elaborate and compress the extracted information towards the final layer of the network. Due to the large number of features extracted in the first layers, the inference time will be relatively high when compared with the other architectures.
- *thin_to_fat*: This topology provides an increasing number of filters at each layer while the feature maps' height and width decrease. The network configuration is 2+2+2. The convolutional layers have 20, 25, 30, 30, 40, and 40 filters, respectively. The architecture puts more emphasis on low-level features as compared to high-level ones. By extracting few features in the first layers keeps the execution time at moderate levels.
- *bell*: A cross of *thin_to_fat* and *fat_to_thin* architectures, the number of features extracted with the *bell* topology follows a "bell-shaped" curve. The network configuration is 2+6+2. The convolutional layers have 10, 10, 20, 20, 25, 25, 25, 30, 30 and 20 filters, respectively.

We collected the execution times of the different CNN alternatives, as shown in Table 4. Execution times are influenced by the architecture itself, designed according to the input size, which is the same for all steps in this case study. To reduce the size of the CNN inputs, and consequently the execution time, a *Resizing Block* is used. We have empirically observed that a 15x15 *Resizing Block*, which performs downsampling by 15 for each dimension of its input, has provided the best possible performance in terms of the accuracy/execution time trade-off.

The CNN performance is evaluated in terms of its ability to discriminate between the usable and unusable images, which depends on the dataset and the choice of the threshold α. For each application step, we analyzed the performance of different CNN architectures. Detailed results for the R&C step are reported in Table 5. The CNN is triggered by the *Difference Evaluator*, configured as previously specified ($\alpha = 7\%$). All alternatives are affected by a limited but a non-zero number of false positives (/D /U). In accordance

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2965518, IEEE Transactions on Computers

Table 5
Performance of different CNN architectures for the R&C step

CNN arch.	D /U	/D U	D U	/D /U
bell	54.64	40.54	4.72	0.10
fat_to_thin	54.32	41.48	3.78	0.42
thin_to_fat	53.93	42.00	3.27	0.81
const_filts	54.22	41.71	3.56	0.52

Table 6
Intermediate Comparative Analysis

Approach	D /U	/D U	D U	/D /U	Avg. Time (ms)
TRC + CNN [5]	50.3%	35.6%	13.9%	0.2%	691.17
DiffEv + CNN	50.4%	39.9%	9.6%	0.1%	672.21

with the design strategy outlined in Section 4.2.1, we select the solution with the smallest number of false negatives. In the case of the R&C step, the corresponding optimal architecture turns out to be the *bell* architecture.

The same study has been carried out on all the filters of the Case Study #1. The *bell* architecture offers the best results with respect to the other alternatives for all steps except the Thresholding (T) step of the process, where the *const_filts* solution performs better.

The next step of the design flow is pruning the chosen CNNs. The pruning is performed in such a way that it will improve the time-efficiency of the CNN without any loss of its classification performance. Note that an alternate design flow for a better time-efficiency/classification performance trade-off could be to choose between the pruned versions of different network architectures. On the other hand, in our experiments, it became apparent that such a design flow will not be feasible as pruning is a computationally-intensive process. Therefore, for each SC, we have first selected the best unpruned network architecture, and then proceeded with pruning the chosen architecture.

Our design approach is comparable to the one presented in [5], where the CNN is triggered by the TRC and is not pruned. Although that previous scheme is only a special case of our final proposed scheme, the design space exploration on the CNN architecture and the use of the *Difference Evaluator* allow us to achieve already a better performance, as highlighted by the results in Table 6, where we report the overall performance of the schemes, and the execution times averaged on the distribution of classification cases (D /U, /D U, D U and /D /U) that occur.

5.2.4 CNN Pruning

We applied the two strategies described in Section 4.2.4, APoZ followed by SoAW. Obviously, at the end of the pruning process, the architecture of the initial CNN is modified. However, for brevity, we refer to the modified architecture with the same name as the design we initially adopt. Table 7 reports the execution times of the CNNs, together with the achieved savings, that range from 69% to 88%.

Once the pruning has been performed, the final implementation of each SC module is completed. Table 8 reports the execution times for the four SC modules of this case study, also listing the same information for the baseline solutions. In particular, the first two columns list the name

Table 7
CNNs Execution Times – Case Study #1

Appl. step	CNN Arch.	Exec. time (ms) Initial	Pruned	Improve.
S	pruned *bell*	18.10	2.52	86%
T	pruned *const_filts*	33.20	10.28	69%
R&C	pruned *bell*	18.10	2.21	88%
A	pruned *bell*	18.10	2.63	85%

Table 8
Checkers' Execution Times – Case Study #1

Appl. step	Exec. time (ms)	TRC (ms)	Worst Case [5] (ms)	Worst Case SC (ms)	Avg. Case [5] (ms)	Avg. Case SC (ms)
S	97.0	4.7	29.8	14.22	29.8	10.80
T	32.0	4.7	29.8	21.98	29.8	17.67
R&C	185.0	4.7	29.8	13.91	29.8	10.29
A	34.0	4.7	29.8	14.33	29.8	9.79

of the application step and its execution time. Columns three and four report the execution times for the TRC adopted in the classical DWC solution, and the checker adopted in [5] (TRC and CNN), respectively. The last column reports the time of the proposed SC when the CNN is invoked. When the CNN is not invoked, only the *Difference Evaluator* is executed, resulting in an execution time that is identical to the execution time of the TRC.

5.2.5 Complete Hardened Application

It is now possible to analyze the complete hardened application, in terms of performance and execution time. We compare the proposed scheme against the baseline schemes, namely the DWC and the scheme proposed in [5] (TRC and CNN) enhanced with the selection of the best CNN architecture, not part of the original work (labeled $[5]^{+}$).

The results of this analysis are reported in Table 9. Pruning yields a 7.12% execution time improvement with respect to the SC with the initial CNN architectures, leading to a final 14.89% improvement with respect to the classical DWC solution. In comparison with the scheme in [5], a 10.36% improvement is obtained.

By looking at the performance of the pruned solution with respect to the usability/discarding of the images, it is worth noting that the pruning does not significantly affect the accuracy, and even reduces the number of false negatives to 0.3%. We believe this may happen as a new training phase is performed after pruning every connection.

The final SC produces an average of 8.5% of D U images, while the amount of /D /U is 0.3%, leading to a final SC accuracy of 91.2%. When considering the reliability of the hardened application, we only take false negatives into account. The end result is an accuracy of 99.7%, which is within acceptable margins in non-safety-critical scenarios.

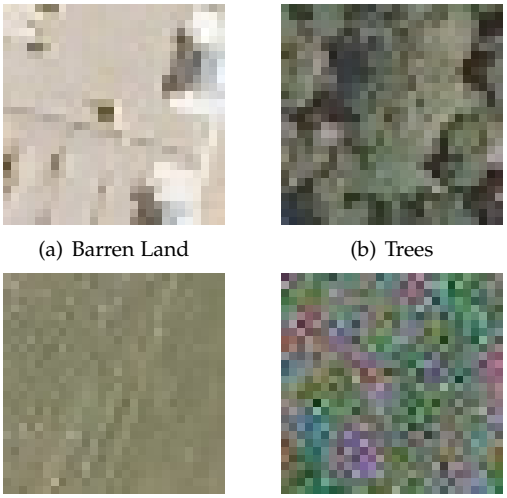
5.3 Case study #2: Land Segmentation

The second application we considered takes as input patches of aerial images and uses a CNN to classify each patch. The CNN outputs the likelihood of the patch to belong to each one of 4 considered classes (barren land,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2965518, IEEE Transactions on Computers

Table 9
Fault Management Scheme Performance – Case Study #1

Approach	D /U	/D U	D U	/D /U	Avg. time (ms)
TRC (classical DWC)	50.5%	0.0%	49.5%	0.0%	733.60
[5]$^{+}$	50.4%	35.5%	14.0%	0.1%	696.53
DiffEv + CNN	50.4%	39.9%	9.6%	0.1%	672.21
DiffEv + pruned CNN	50.2%	41.0%	8.5%	0.3%	624.33



(a) Barren Land (b) Trees

(c) Grassland (d) Other

Figure 6. Examples of classes for Case Study #2. Images are taken from SAT-4 database [36].

trees, grassland, or other). As the classes are mutually exclusive, all the likelihoods sum up to 1 and therefore the final classification is the class with the highest likelihood. Fig. 6 shows some examples of the patches in the dataset with the respective classes reported in the caption. To structure the application in multiple steps, we split the land segmentation application into a number of intermediate layers. The first layer accepts an image and produces a feature matrix. The intermediate layers accept a feature matrix and produce a feature matrix. Finally, the final layer accepts a feature matrix and produces the final classification.

Again, as a simple end-user application for the land segmentation we implemented an oracle that given a (possibly faulty) output o_f and the corresponding golden counterpart o_g assigns to the result a *usable* label if the Manhattan distance (or L1-norm) of the two output vectors $\|o_f - o_g\|_1$ is lower than a set threshold 0.05.

5.3.1 Training/Validation/Test Set Generation

For the second case study we used as input dataset a subset of 20,000 images from the SAT-4 database [36].

We performed fault injection experiments on the various layers and the outcome is the following set of error models: i) *matrix shift*, a part of the feature matrix is shifted; ii) *zero area*, a portion of the feature matrix is set to zeros; iii) *zero spots*, pixels, lines or squares of the feature matrix are set to zeros; iv) *value change*, the values of a portion of the feature matrix are increased/decreased of a given constant value.

We corrupted each one of the input images by simulating 10 different error models, generating a training set of 200,000 corrupted images. Similarly, we generated the validation and test sets starting from 6,000 and 10,000 different images, respectively, taken from the same database.

Table 10
Execution times of different SCs architectures – Case Study #2

	Exec. time (ms)			
	1 Branch	*2 Branches*	*4 Branches*	*8 Branches*
Step 1	10.10	11.58	23.68	47.50
Step 2	4.60	3.01	6.10	12.25
Step 3	1.10	1.08	2.20	4.55

Table 11
Performance of different CNN architectures for Step 1 – Case Study #2

	D /U	/D U	D U	/D /U
1 Branch	14.92	80.99	3.14	0.95
2 Branches	12.90	82.97	1.16	2.98
4 Branches	13.39	81.86	2.27	2.48
8 Branches	12.55	82.72	1.41	3.33

5.3.2 α Threshold Computation

As for the first application, the first step here is to determine the α thresholds for the *Difference Evaluator*s in the SCs. To this purpose we ran the same experiment reported in Section 5.2.2. This experiment highlighted that for the second application there is always a non-zero number of false negatives, even for very low threshold values. Thus, to avoid misclassifications, it is necessary to set $\alpha = 0\%$, that is to use a TRC for all three steps of the application.

5.3.3 CNN Architecture Design and Training

The spatial dimensions of the output of the three application steps are relatively small, ranging from 5x5 to 26x26. Thus, the CNN of the SC does not need to be particularly deep for accurate image classification and input resizing is not necessary. We thus considered a "branching" architecture, where n different branches of CNNs accept the same input but operate independently. In other words, the branches do not share any weights or connections. The outputs of the n CNN branches are concatenated and fed to a final fully-connected layer, which provides the classifier output. We explored four CNN architectures, characterized by 1, 2, 4 and 8 branches (the first architecture being equivalent to a "regular" CNN). We report the resulting execution times in Table 10. Since the size of the inputs of each CNNs is different between the three application steps, we observe different execution times.

The CNN classification performance for each possible architecture is reported in Table 11 for Step 1 of the application. As before, for each application step, we select the architecture characterized by the smallest number of /D /U cases. The corresponding best architecture is "*1 Branch*" for Step 1. The same choice is carried out for Step 3, in which case the "*2 Branches*" architecture is selected. In Table 12, we compare our unpruned intermediate solutions against the scheme proposed in [5]. The results highlight the benefits of exploring the solution space with respect to the CNN architecture, since both solutions use the TRC to trigger classification.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2965518, IEEE Transactions on Computers

Table 12
Intermediate Comparative Analysis – Case Study#2

Approach	D /U	/D U	D U	/D /U	Avg. time (ms)
TRC + CNN [5]	11.9%	75.7%	9.6%	2.8%	137.58
TRC + CNN	12.4%	76.4%	8.9%	2.3%	135.85

Table 13
CNNs Architecture Design and Execution Times – Case Study #2

Appl. step	CNN Arch.	Exec. time (ms) Initial	Pruned	Improve.
Step 1	*1 Branch*	10.10	4.25	58%
Step 2	*2 Branches*	3.01	1.50	50%
Step 3	*1 Branch*	1.10	0.60	45%

Table 14
Checkers' Execution Times – Case Study #2

Appl. step	Exec. time (ms)	TRC (ms)	[5] (ms)	SC (ms)
Step 1	30.25	0.09	11.65	4.34
Step 2	42.65	0.04	3.05	1.54
Step 3	26.0	0.01	1.09	0.61

Table 15
Fault Management Scheme Performance – Case Study #2

Approach	D /U	/D U	D U /D /U		Avg. time (ms)
TRC	14.7%	0.0%	85.3%	0.0%	197.94
TRC + CNN [5]$^{+}$	12.4%	76.4%	8.9%	2.3%	135.85
TRC + Pruned CNN	13.1%	74.9%	10.4%	1.6%	129.20

5.3.4 CNN Pruning

We applied the same two pruning strategies described in Section 4.2.4 and re-trained the CNNs. The resulting execution times are reported in Table 13. We can observe that pruning provides a 45%-58% improvement in execution times. The complete execution times of the various checker solutions are shown in Table 14. We note the significant improvement compared to our initial work [5].

5.3.5 Complete Hardened Application

Execution times for the implemented SC modules are used to determine the performance of the completely hardened architecture. We report the overall performance and execution times in Table 15, and compare with the baseline solutions. Similar to the previous case study, the number of false negatives provided by the pruned networks is lower than the unpruned networks. The final implementation is characterized by an accuracy of 88.00%. When computing the reliability of the solution, affected by false negatives only, we achieve a 98.40%. From the execution time point of view, the final implementation improves the classical DWC solution of 34.72%, and the proposal in [5] by 4.89%.

As a final consideration, both the actual and the ideally optimal solutions are characterized by higher improvements, notably because of the nature of the adopted application, that is inherently more tolerant to a certain degree of inexactness.

6 Conclusions

This paper has presented a novel time-efficient flexible fault management scheme for image processing applications. We extend the classical DWC approach by predicting the usability of the application output, based on a classification of the intermediate, possibly corrupted, data. Such fault management scheme stems from the adoption of a new paradigm in fault detection/tolerance: we move from the classical corrupted/not corrupted image model to usable/unusable one. The result empowers image processing applications with fault tolerance requirements to discard only those outputs that would actually lead to a system failure. Avoiding unnecessary re-executions reduces the typical reliability-related overheads, such as time and power.

The proposed approach has been applied to two image processing applications related to the aerospace scenario. Experimental results have demonstrated the effectiveness of the approach, supported by the overall time saving of 14.89% and 34.72% for the two applications, respectively. Our approach has also incurred a low rate of false negatives, being corrupted images that could not be detected. In particular, the false negative rates have been in the range of 0.3% and 1.6% for the two case studies, respectively, and thus are well within acceptable margins.

References

[1] R. L. Davidson and C. P. Bridges, "Error Resilient GPU Accelerated Image Processing for Space Applications," *IEEE Trans. Parallel and Distributed Systems*, vol. 29, no. 9, pp. 1990–2003, 2018.

[2] S. Yin, B. Xiao, S. X. Ding, and D. Zhou, "A Review on Recent Development of Spacecraft Attitude Fault Tolerant Control System," *IEEE Trans. Industrial Electronics*, vol. 63, no. 5, pp. 3311–3320, 2016.

[3] M. A. Hoffmeyer, K. Miller, I. Balter, and T. J. Roh, "Satellite communications data processing," 2018, US Patent 9,954,602.

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

[5] M. Biasielli, C. Bolchini, L. Cassano, and A. Miele, "A Smart Fault Detection Scheme for Reliable Image Processing Applications," in *Proc. Design, Automation and Test in Europe Conf.*, 2019, pp. 698–703.

[6] I. Albandes, A. Serrano-Cases, A. J. Sánchez-Clemente, M. Martins, A. Martínez-Álvarez, S. Cuenca-Asensi, and F. L. Kastensmidt, "Improving approximate-TMR using multi-objective optimization genetic algorithm," in *Proc. Latin-American Test Symp.*, 2018, pp. 1–6.

[7] A. J. Sanchez-Clemente, L. Entrena, R. Hrbacek, and L. Sekanina, "Error Mitigation Using Approximate Logic Circuits: A Comparison of Probabilistic and Evolutionary Approaches," *IEEE Trans. on Reliability*, vol. 65, no. 4, pp. 1871–1883, 2016.

[8] I. A. C. Gomes, M. G. A. Martins, A. I. Reis, and F. Lima Kastensmidt, "Exploring the use of approximate TMR to mask transient faults in logic with low area overhead," *Microelectronics Reliability*, vol. 55, no. 9, pp. 2072–2076, 2015.

[9] M. R. Choudhury and K. Mohanram, "Approximate logic circuits for low overhead, non-intrusive concurrent error detection," in *Proc. Design, Automation & Test in Europe Conf.*, 2008, pp. 903–908.

[10] K. Chen, J. Han, and F. Lombardi, "Two Approximate Voting Schemes for Reliable Computing," *IEEE Trans.Computers*, vol. 66, no. 7, pp. 1227–1239, 2017.

[11] B. Shim, S. R. Sridhara, and N. R. Shanbhag, "Reliable low-power digital signal processing via reduced precision redundancy," *IEEE Trans. Very Large Scale Integration Systems*, vol. 12, no. 5, pp. 497–510, 2004.

[12] A. Ullah, P. Reviriego, S. Pontarelli, and J. A. Maestro, "Majority voting-based reduced precision redundancy adders," *IEEE Trans. Device and Materials Reliability*, vol. 18, no. 1, pp. 122–124, 2018.

[13] P. J. Eibl, A. D. Cook, and D. J. Sorin, "Reduced Precision Checking for a Floating Point Adder," in *Proc. Symp. Defect and Fault Tolerance in VLSI Systems*, 2009, pp. 145–152.

[14] R. Hegde and N. R. Shanbhag, "Soft digital signal processing," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 6, pp. 813–823, Dec 2001.

[15] B. Shim and N. R. Shanbhag, "Energy-efficient soft error-tolerant digital signal processing," *IEEE Trans. Very Large Scale Integration Systems*, vol. 14, no. 4, pp. 336–348, 2006.

[16] E. Koyuncu and H. Jafarkhani, "Variable-length limited feedback beamforming in multiple-antenna fading channels," *IEEE Trans. Information Theory*, vol. 60, no. 11, pp. 7140–7165, Nov 2014.

[17] E. Koyuncu, X. Zou, and H. Jafarkhani, "Interleaving channel estimation and limited feedback for point-to-point systems with a large number of transmit antennas," *IEEE Trans. Wireless Communications*, vol. 17, no. 10, pp. 6762–6774, Oct 2018.

[18] Microsoft Corporation, "Bing Maps APIs," https://msdn.microsoft.com/en-us/library/dd877180.aspx, 2018.

[19] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults," in *Proc. Conf. Dependable Systems and Networks*, 2014, pp. 375–382.

[20] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design," in *Proc. European Conf. Computer Vision*, 2018, pp. 122–138.

[21] Y.-L. Boureau, J. Ponce, and Y. LeCun, "A Theoretical Analysis of Feature Pooling in Visual Recognition," in *Proc. Conf. Machine Learning*, 2010, pp. 111–118.

[22] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, and G. Wang, "Recent Advances in Convolutional Neural Networks," *Pattern Recognition*, vol. 77, pp. 354–377, 2018.

[23] C.-Y. Lee, P. W. Gallagher, and Z. Tu, "Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree," in *Artificial Intelligence and Statistics*, 2016, pp. 464–472.

[24] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[25] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," in *Proc. Conf. Learn. Representations*, 2015, pp. 1–13.

[26] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[27] G. Hinton, "Overview of mini-batch gradient descent," http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[28] R. Caruana, S. Lawrence, and C. L. Giles, "Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping," in *Advances Neural Information Processing Sys.*, 2001, pp. 402–408.

[29] S. M. Kakade, S. Shalev-Shwartz, and A. Tewari, "Regularization techniques for learning with matrices," *Journal Machine Learning Research*, vol. 13, no. Jun, pp. 1865–1890, 2012.

[30] P. Panda, A. Sengupta, and K. Roy, "Conditional deep learning for energy-efficient and enhanced pattern recognition," in *Proc. Conf. Design, Automation & Test in Europe*, 2016, pp. 475–480.

[31] D. Stamoulis, T.-W. R. Chin, A. K. Prakash, H. Fang, S. Sajja, M. Bognar, and D. Marculescu, "Designing adaptive neural networks for energy-constrained image classification," in *Proc. Conf. Computer-Aided Design*, 2018, pp. 23:1–23:8.

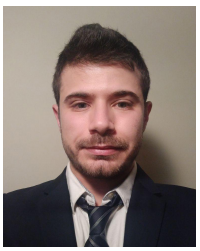
[32] J.-H. Luo, J. Wu, and W. Lin, "ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression," in *Proc. IEEE Conf. Computer Vision*, 2017, pp. 5068–5076.

[33] M. Abadi *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015. [Online]. Available: www.tensorflow.org/

[34] M. Tan, B. Chen, R. Pang, V. Vasudevan, and Q. V. Le, "MnasNet: Platform-Aware Neural Architecture Search for Mobile," in *Proc. Computer Vision and Pattern Recognition Conf.*, 2018.

[35] G. Bender, P.-J. Kindermans, B. Zoph, V. Vasudevan, and Q. Le, "Understanding and simplifying one-shot architecture search," in *Proc. Conf. Machine Learning*, 2018, pp. 549–558.

[36] S. Basu, S. Ganguly, S. Mukhopadhyay, R. DiBiano, M. Karki, and R. Nemani, "Deepsat: a learning framework for satellite imagery," in *Proc. SIGSPATIAL Conf. Advances in Geographic Information Systems*, 2015.

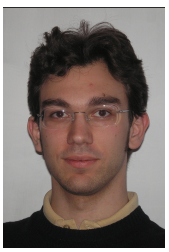


Matteo Biasielli received his M.S. from University of Illinois at Chicago and from Politecnico di Milano in 2019 in Computer Science Engineering. His interests are in the area of Artificial Intelligence and Machine Learning, working on solutions for fault tolerance in image processing applications.

Cristiana Bolchini is a Professor at Politecnico di Milano, where she received a Ph.D. in Automation and Computer Engineering in 1997. Her research interests cover the areas of methodologies for the design and analysis of digital systems with a specific focus on dependability and self-awareness for heterogeneous system architectures. She has co-authored more than 125 papers in peer-reviewed international journals and conferences. In 2019, she is serving as the Technical Programme Chair of the conference IEEE/ACM Design Automation and Test in Europe.

Luca Cassano is an Assistant Professor at Politecnico di Milano, Italy. He received the B.S., M.S. and Ph.D. degrees in Computer Engineering from the University of Pisa, Italy. His research activity focuses on the definition of innovative techniques for fault simulation, testing, untestability analysis, diagnosis, and verification of fault tolerant and secure digital circuits and systems. With his Ph.D. thesis, titled "Analysis and Test of the Effects of Single Event Upsets Affecting the Configuration Memory of SRAM-based FPGAs", he won the European semifinals of the 2014 TTTC's E. J. McCluskey Doctoral Thesis Award and he classified as runner-up at the world finals.

Erdem Koyuncu received the B.S. degree from the Department of Electrical and Electronics Engineering, Bilkent University, in 2005, and the M.S. and Ph.D. degrees from the Department of Electrical Engineering and Computer Science, University of California at Irvine in 2006 and 2010, respectively. From 2011 to 2016, he was a Post-Doctoral Scholar at the Center for Pervasive Communications and Computing at the same institution. From 2016 to 2018, he was a Research Assistant Professor at the Department of Electrical and Computer Engineering, University of Illinois at Chicago, where he is currently an Assistant Professor.



Antonio Miele is an Assistant Professor at Politecnico di Milano since 2014. He holds a M.S. in Computer Engineering from Politecnico di Milano and a M.S. in Computer Science from the University of Illinois at Chicago. In 2010 he received a Ph.D. degree in Information Technology from Politecnico di Milano. His main research interests are related to design methodologies for embedded systems, in particular fault tolerance and reliability issues, runtime resource management in heterogeneous multi-/many-core systems and FPGA-based systems design. He is co-author of more than 70 peer-reviewed publications.