POLITECNICO DI TORINO
Repository ISTITUZIONALE

RePAIR: Reconfigurable Platform for AI Resilience within RISC-V Ecosystem

Original
RePAIR: Reconfigurable Platform for AI Resilience within RISC-V Ecosystem / Cora, G., Vacca, E., De Sio, C., Azimi, S., Sterpone, L.. - 15594:(2025), pp. 71-87. (21st International Symposium on Applied Reconfigurable Computing ARC 2025 Seville (ESP) April 9–11, 2025) [10.1007/978-3-031-87995-1_5].

Availability:
This version is available at: 11583/2998583 since: 2025-09-01T08:39:31Z

Publisher:
Springer

Published
DOI:10.1007/978-3-031-87995-1_5

Terms of use:

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

Publisher copyright
Springer postprint/Author's Accepted Manuscript

This version of the article has been accepted for publication, after peer review (when applicable) and is subject to Springer Nature's AM terms of use, but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/10.1007/978-3-031-87995-1_5

(Article begins on next page)

13 September 2026

RePAIR: Reconfigurable Platform for AI Resilience within RISC-V Ecosystem

Giorgio Cora[1[0009-0008-3720-0379]], Eleonora Vacca[1[0000-0002-7573-1815]], Corrado De Sio[1[0000-0003-4212-3052]], Sarah Azimi[1[0000-0002-9169-6140]], Luca Sterpone[1[0000-0002-3080-2560]]
[1] Politecnico di Torino, Torino, Italy
giorgio.cora@polito.it, eleonora.vacca@polito.it, corrado.desio@polito.it, sarah.azimi@polito.it, luca.sterpone@polito.it

Abstract. Recently, platforms combining RISC-V processors with accelerators for deep-learning applications have gained popularity even for high-reliability applications such as avionics and space. However, for high-performance safety-critical systems, it is mandatory to couple high-performance architecture with reliable mechanisms for coping with errors and faults. We propose the first FPGA-based architecture that combines a RISC-V processor with a systolic array-based accelerator, a fault detection, fault correction, and an execution recovery mechanism. The proposed solution corrects faults in the systolic array datapath by exploiting a partial reconfiguration mechanism. When an error is detected, the RISC-V processor can trigger the accelerator reconfiguration, correcting the fault. Furthermore, the approach allows resuming the inference from the last correctly executed step, significantly reducing the availability overhead. The approach results in a high-performance and high-reliable platform that can autonomously detect and correct faults, providing execution continuity and minimal system downtime.

Keywords: RISC-V, Systolic Arrays, Fault Detection, Partial Reconfiguration, AI.

1 Introduction

The widespread adoption of Deep Learning (DL) techniques, combined with the increase in model complexity, has driven the development of high-performance platforms capable of delivering the computational power required for efficient and rapid inference. Due to the open-source nature of the RISC-V Instruction Set Architecture (ISA), researchers are increasingly focusing on enhancing RISC-V-based solutions to meet these computational demands. The predominant strategies involve either extending the ISA [1] or coupling RISC-V with application-specific accelerators [2]. Since most DL workloads involve large-scale matrix multiplications, many proposed solutions integrate methods for executing neural networks, such as general matrix multiplications (GEMM), within the RISC-V environment [3][4], which has renewed interest in Systolic Array (SA) architectures. Solutions based on SA employ a grid-like structure of processing elements (PEs) for efficient parallel computation, minimizing memory access, maximizing data reuse, and performing on-chip computation. These aspects are

appealing for deep neural networks (DNNs)-based applications, where minimizing data movement becomes a priority to achieve high computation efficiency. Due to the complexity of DNN, modern SA-based accelerators, with Google's Tensor Processing Unit being a pioneering example [5], are equipped with their own ISA supporting single-instruction multiple data execution to process large vectors of data in one clock cycle.
However, when DNN models are employed in safety-critical applications, focusing only on computational efficiency is insufficient. Indeed, ensuring the correctness of the inference execution is as important as the performance. For this reason, several works proposed methodologies to address hardware fault detection in the SA accelerator [6] or fault masking acting on the DNN model [7][8]. Still, there's a lack of solutions that target reconfigurable platforms, such as FPGA, when adopted to implement such kinds of AI accelerators.
The demand for fault-tolerance solutions for SRAM-based FPGAs is also exacerbated by the fact that they are susceptible to errors in configuration memory caused by external factors, such as ionizing radiation, which is the primary concern for space and avionics systems and can also occur at sea level [9]. A trivial way to address hardware faults in the implemented circuit due to configuration memory (CRAM) corruption is to reload the configuration data (CDATA) to restore the correct circuit functionalities using the unfaulty configuration bitstream. However, this approach has substantial limitations since it takes several seconds, which results in a reduction of system availability. Moreover, runtime fault detection in FPGA devices is often invasive to applications and computational systems. Standard fault detection methodologies may require additional modules allocated along with the implemented circuit, like SEM-IP [10], or to perform periodic memory readback to compare with a golden configuration, eventually triggering complete reconfiguration upon error detection. The former approach also diminishes the resources dedicated to computational units, and additionally, correcting the error does not consider errors already propagated in the user logic between the error and the correction. The latter fails to meet the real-time constraints of DNN, as it can take several seconds to detect and correct an error—potentially leading to catastrophic failures or service disruption in safety-critical systems.

1.1 Main Contributions

This work presents RePAIR, a heterogeneous computing platform integrating and extending an open-source RISC-V processor[11] and an open-source TPU-like accelerator [12]. RePAIR targets reconfigurable devices and offers fault detection, fault correction, and recovery capability, preserving high performance and minimum overhead. To the best of our knowledge, this is the first platform to combine performance optimization and fault tolerance in this manner. The main innovations and contributions are as follows:

- Design and implementation of the RePAIR platform, integrating a RISC-V processor with a systolic array-based accelerator.
- Extension of tinyTPU ISA [12] for Runtime Self-Test: The ISA of the AI accelerator is extended to enable runtime self-test capabilities. This feature allows the detection of structural faults in the accelerator datapath with minimal impact on the inference process.

- Dual Inference Modes: the computational platform can execute either in a *plain mode* for standard inference operation without additional overhead or in *testing mode*, which introduces a checksum test for the current inference workload at the cost of three additional clock cycles per matrix multiplication. This mechanism enables fault detection during inference without disrupting the normal operational flow with an accelerator area overhead of only 0.31%.
- Fault Detection, Correction, and Recovery: When a fault is detected in the TPU datapath, the accelerator notifies the RISC-V processor. The processor triggers a dynamic partial reconfiguration to reload the bitstream section associated with the faulty accelerator while preserving its ongoing correct workload. By utilizing partial reconfiguration, the RePAIR platform implemented on the AMD KCU105 device can reduce system downtime by up to 900 times compared to traditional methods. This ensures rapid fault recovery and enhances system availability. The recovery mechanism resumes the workload from the last correctly executed instructions. This approach limits the inference execution overhead to 30% in the worst-case scenario, compared to full device reconfiguration, which can result in up to 96% overhead.

The content of the paper is structured as follows. Section II introduces the related works. Section III details the SA fault detection mechanism, while section IV outlines the integration of the RISC-V processor with the accelerator. Finally, section V presents the experimental results, and Section VI concludes.

2 State of the Art

Over the years, the introduction of the RISC-V ISA has led to the development of various computing platforms [13], addressing the needs of many domains of targeting DNN applications [14][16], with remarkable results. For instance, authors in [3] propose a custom, high-performance, and multithread library for convolutional operations targeting RISC-V processors. Results show extremely good performance in terms of Floating-Point Operations Per Second (FLOPS), even when compared to other processors such as ARM ones, proving their suitability for carrying out DNN operations. One of the main characteristics of RISC-V processors is the open-source nature of the ISA, allowing for easier pipeline modification to support custom instruction execution and efficient coupling with external modules. Following this trend, authors in [17] propose a custom architecture that couples a 64-bit RISC-V architecture, Ariane, with a co-processor for DNN inference applications. The possibility of extending the ISA of the RISC grants faster and more efficient DDR access, improving the overall execution time. Authors in [4] developed an architecture that embeds custom modules and co-processors in the RI5CY processor pipeline to support GEMM operations. The acceleration of GEMM operation is achieved through the ISA extension, which provides three custom SIMD instructions to be executed depending on the selected parallelism. Similarly, authors in [18] propose and compare two similar designs, evaluating the differences in terms of execution speedup when co-processors are instantiated inside or

outside the RISC-V pipeline. In the first approach, the co-processor is coupled with the 64-bit Rocket architecture through the TileLink Bus, while in the latter solution, the Matrix multiply unit is directly instantiated inside the soft-processor pipeline. Results show a speedup of 1.3x for the latter method with respect to the external coupling, while both grant improved performances against the plain version of Rocket architecture.

However, these RISCV-based platforms solely focus on performance, leaving a gap in the adoption of DNN reliability in a critical domain. We fill such a gap by proposing a platform combining a RISCV processor and a TPU accelerator oriented to enhance the system's dependability by combining three features: an error detection technique based on checksum, a fault correction mechanism based on FPGA reconfiguration capability, and an execution recovery mechanism for resuming execution from the last correct computation. In our proposed approach, we assure runtime error detection capabilities in the DNN accelerator by extending the ISA of a TPU coupled with a processor system with custom instructions that detect and notify faults. We do not only notify the RISC-V processor system that a fault is affecting the datapath, but we also trigger a dynamic partial reconfiguration of the accelerator to correct the soft errors and recover the DNN inference from the last correct operation. As a result, we proposed the RePAIR platform that targets performance and reliability with a minimal performance penalty and resource overhead.

3 The Proposed Error Detection Mechanism

3.1 Fault Model

The proposed methodology introduces a novel approach for detecting computational errors caused by structural faults in the datapath of systolic arrays. When considering designs implemented in reconfigurable computing platforms, these faults typically arise from the corruption of CRAM —a common issue in harsh environments like space [19], further exacerbated by increased sensitivity as technology scales down[20]. Radiation-induced errors in the CRAM of a reconfigurable system, such as FPGA, may introduce errors corrupting the hardware design implementation. Since CRAM is usually written only during the system boot, errors accumulating in the CRAM will cause errors and eventually lead to system failure [21][22].

3.2 Limitation of Traditional Approaches

Traditional reliability enhancement techniques adopted in such kinds of platforms based on hardware and/or software redundancy impose significant overheads on computation, memory, and energy, making them impractical for DNN applications. For instance, while AMD's Soft Error Mitigation (SEM) IP can detect and restore CRAM faults in runtime, the detection latency, ranging from 25 ms for Kintex-7 to 13 ms for KU040 [23] devices, is incompatible with the real-time detection demands of DNN inference. Furthermore, SEM IP is resource-intensive, consuming up to 835 LUTs, 506 FFs, and multiple BRAM blocks, reducing resources available for application-specific designs. Another aspect to consider is that using SEM-IP provokes operational frequency design constraints. Indeed, the vendor datasheet imposes a max frequency of

100 MHz, which is incompatible with high-performance computing platforms [23]. Finally, correcting configuration memory will not correct previously erroneous generated output or faulty state, creating transient errors that could manifest unexpectedly in the future. Another potential scenario is full device reconfiguration, which does not require additional hardware resources but has two significant drawbacks. First, since the FPGA reboots during this process, the program restarts from the beginning, resulting in the loss of all previously completed computations, therefore increasing the inference execution overhead. Second, full device reconfiguration is time-intensive, taking several seconds to complete—a duration that grows with the device size—substantially prolonging system downtime. Considering KCU105 device, the full board reconfiguration takes 13 seconds while moving to a smaller device as the Ultrascale+ PYNQ-ZU, it requires 6.9 seconds.

3.3 Proposed Approach

Our proposed method addresses issues that make traditional approaches unsuitable for high-performance safety-critical systems, particularly in ensuring reliable execution of DNN inference. It significantly reduces the resource overhead by exploiting the computational resources already existing within the accelerator datapath. Furthermore, it is specifically designed to enable rapid fault detection and recovery during runtime, ensuring compatibility with real-time performance requirements, exploiting dynamic partial reconfiguration for error correction, and the RISC-V for enabling the TPU computational flow recovery.

The methodology combines key elements of ABFT for systolic arrays and scan chain techniques. Similar to systolic array-oriented ABFT [6][24], the proposed method detects faults by computing checksum values on the data processed by the accelerator, but without using additional hardware resources. Indeed, drawing on the scan chain approach [25][26], the checksums are computed by propagating specific test patterns through the functional path of the systolic array.

In detail, let's consider the normal operation executed by the processing element $PE_{i,j}$ of a SA grid. After the neural network weight loading in the PEs grid, each $PE_{i,j}$ performs a Multiply-and-Accumulate (MAC) operation. Specifically, it multiplies one input data for the weight value $w_{i,j}$ stored in its weight register and accumulates the partial product produced with that coming from the $PE_{i-1,j}$. In the next clock cycle, a new input data feeds the $PE_{i,j}$, while the previous input data is propagated to the $PE_{i,j+1}$. To summarize, we have input data propagation from left edge to the right edge of the array, and partial product accumulation that propagates from top to bottom in each SA column, while weights stay fixed on the PEs.

Our proposed approach exploits these functional paths to propagate test patterns across the PEs. These test patterns are chosen to produce a checksum value and its complement of the weights currently loaded in each column of SA. Hence, base and complemented checksums are produced in consecutive clock cycles. They enable to distinguish between transient faults or soft errors and structural faults induced by CRAM corruption. Such distinction is important since correcting the source of the error for the former only requires re-loading the weight matrix in the SA, and a fresh

execution, so correction is made at the software level. In contrast, the structural faults require CRAM refresh with the unfaulty CDATA.

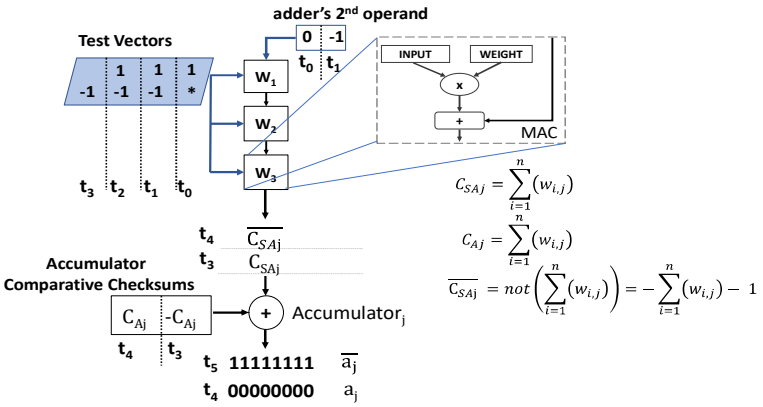


Fig. 1. Dataflow of the proposed fault detection mechanism considering the j column of MACs and the j Accumulator .

Going into detail on the checksum generation shown in Fig. 1, two test patterns are applied as input vectors to the SA in two consecutive clock cycles. These test vectors have *n* elements when considering a PE grid of size *n* x *m*. These test vectors are built to produce checksum values C_{SAj} , corresponding to the sum of the weights loaded in the processing grid at column *j,* and its complementary value $\overline{C_{SAj}}$ as reported in Eq. 1 and Eq. 2. To compute C_{SAj} it is sufficient to have all the elements of the first vector equal to 1. In contrast, to produce $\overline{C_{SAj}}$ all elements of the second vector are equal to -1, and the second operand of the adders of the SA first MACs row is fed with -1.

$$C_{SAj} = \sum_{i=1}^{n}(w_{i,j}) \qquad \text{with j } 0, 1\ldots m \quad (1)$$

$$\overline{C_{SAj}} = not\left(\sum_{i=1}^{n}(w_{i,j})\right) = -\sum_{i=1}^{n}(w_{i,j}) - 1 \qquad \text{with j } 0, 1\ldots m \quad (2)$$

Since the two test vectors used for checksum computation have their LSB set to 1, an additional vector of 0s must be propagated to evaluate the flipping of this bit. This ensures that no stuck-at-1 fault affects it. The SA will produce all 0s in case no fault affects this bit. For the ABFT approach to be effective, we must compare the generated checksums to those produced by another unit to detect any mismatch signaling a possible error. Commonly, TPU architecture is equipped with an external bank of accumulators adjacent to the MAC grid to support tiling when the matrix multiplication does not fit in with the available resources. Our proposed approach exploits these external accumulators to compute the comparative checksum value. To do so, when the weights are loaded in the SA one vector at a time, they will also flow through the accumulators, which sum the weights data, generating C_{Aj} , as in Eq. 3. The comparison between the checksum values produced by the SA and those produced by the accumulators will happen by exploiting the accumulators themselves.

$$C_{Aj} = \sum_{i=1}^{n}(w_{i,j}) \quad (3)$$

As soon as the SA checksums are produced, they are accumulated with those produced by the accumulators, following the traditional datapath of the accelerator. Hence, what changes with respect to the canonical use of the accumulation process is only the meaning of the operands. Indeed, these will not be the results of consecutive matrix multiplications to be merged but checksums values accumulated. Specifically, consider C_{SAj} and $\overline{C_{SAj}}$ as checksum values direct and complemented produced by the column *j* of the SA, and C_{Aj} as the one produced by the accumulator *j*. Each accumulator A_j will perform the following operation:

$$a_j = C_{SAj} - C_{Aj} \quad (4)$$

$$a_j^* = \overline{C_{SAj}} + C_{Aj} \quad (5)$$

In the absence of faults, regardless of the weights' values, each a_j will assume value 0, and each $\overline{a_j}$ the value -1 (i.e. all 1s in binary). By examining the values assumed by each pair of $(a_j, \overline{a_j})$, for each column of PEs, we can identify and distinguish either the presence of a bitflip in a weight register when the pair a_j, a_j^* differs from the fault-free values, and values are complementary or a structural fault. Specifically if $a_j, \overline{a_j}$ differs from the fault-free values and are not complementary, then we have a structural fault in datapath. To identify fault location, we need to look at the checksums produced by the SA. Specifically, if the pair $a_j, \overline{a_j}$ is not complementary while C_{SAj} and $\overline{C_{SAj}}$ are complementary, then the structural fault is located in the *j* accumulator. Otherwise, the fault is located in the *j* column of the SA.

With respect to previous works that required N adders [1], and 2N+1 adders + 1 MAC [24] to compute checksum values in an SA of size N x N, our proposed approach does not require any additional resources while inducing a penalty of just 3 clock cycles to process the test vectors.

4 The RePAIR Platform

The RePAIR platform is based on a pair of open-source computing cores: the NEORV32 RISC-V processor [11] and the tinyTPU accelerator [12]. In the context of DNN applications, the platform implements a traditional processor-coprocessor paradigm, where the NEORV32 orchestrates the execution by supplying the tinyTPU accelerator with a microcode transmitted via the accelerator's instruction FIFO to perform computation layer-by-layer. Upon completing the computation of a layer, NEORV32 retrieves the results and reforms the data to suit the requirements of the subsequent layer. Each layer execution can be conducted in a testing mode, employing the fault detection methodology described in subsection 3.3. If a fault is detected in the tinyTPU datapath, the NEORV32 triggers and manages the dynamic partial reconfiguration of the accelerator to correct the faulty datapath. Once the accelerator is operational again, NEORV32 resumes computation from the last successful execution. This mechanism ensures continuity of application execution while minimizing system downtime.

4.1 Proposed Platform

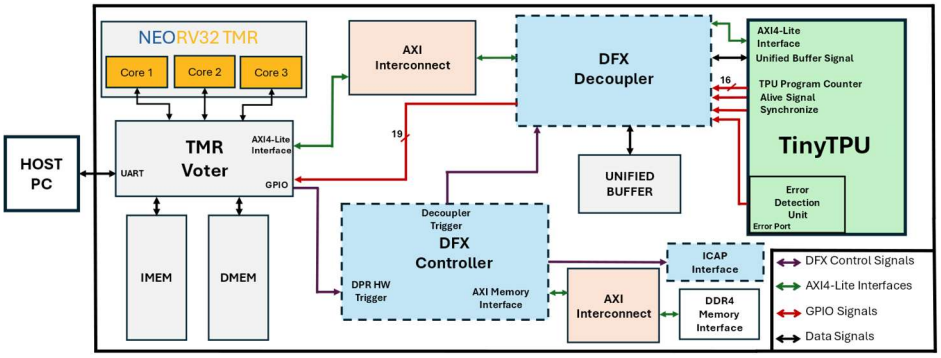


Fig. 2. The RePAIR platform.

Figure 2 provides a schematic view of the implemented architecture and the internal connections, including the UART connection that can be used for interfacing with an external host PC. The data exchange between the processor and the accelerator is handled through AXI interfacing and memory. The TPU can notify fault detection through interruptions, and the TPU status is provided to NEORV using GPIOs. Upon detecting a fault, the NEORV is responsible for initiating the dynamic partial reconfiguration of the accelerator. The dynamic partial reconfiguration [27] is handled by controlling the DFX modules. In particular, the DFX Controller [28] oversees the reconfiguration itself, fetching the CDATA from DDR and writing configuration memory. A DFX decoupler between the TPU and the rest of the system ensures that the interface signals are kept at a stable low value while partial reconfiguration is performed. Finally, if the partial reconfiguration of the TPU fails to correct the error in the datapath, a full FPGA reconfiguration is triggered as a last effort to restore functionality.

4.2 NEORV32

The NEORV32 [11] is an open-source RISC-V processor based on a 4-stage pipelined multi-cycle architecture with a maximum operating frequency of 250MHz supporting the base RV32I ISA. It is highly customizable with various additional modules and extensions. To include NEORV32 in the RePAIR platform, the AXI-Lite communication to interface the system with the TPU has been enabled. Moreover, the UART module has been enabled to support debugging and communication with a host PC. No ALU extensions have been enabled for our platform since most of the computational work in DNN inference is carried out by the TPU accelerator, which performs matrix multiplications and applies activation functions. Since the platform's main goal is to provide performance and highly reliable execution, the NEORV processor has been hardened by design, applying TMR.

4.3 tinyTPU

The tinyTPU [12] is an open-source, VHDL-based machine learning co-processor resembling the behavior and architecture of Google's TPU version 1. It consists of a

parametric design, where the size of a systolic array core can be modified from a minimum size of 6 x 6 MAC units to a maximum size of 14 x 14. Along with SA, the accelerator is equipped with an external accumulator bank. Since it is implemented to execute DNN, it embeds hardwired quantized activation units supporting the ReLu and Sigmoid activation functions. Additionally, it comes with its own weight and input/output buffer where DNN weights and input images/output of each layer are stored. The tinyTPU is a co-processor offering a custom ISA designed for 80-bit instruction parallelism. This instruction format integrates multiple elements, including opcode, memory addresses for source and destination operands, and a field specifying the number of data vectors to be processed in a single instruction. The instructions used to perform DNN inference can be summarized as *read_weights*, *matrix_multiply*, and *activate.*

4.4 Recovery Routine

A new version of the *read_weight* and *matrix_multiply* instructions*, t_read_weight,* and *t_matrix_multiply,* respectively, has been added to the TPU ISA to support the fault detection methodology. The *t_read_weight* instruction performs two tasks. First, it loads weight values from memory into the SA. Second, it propagates these weights through the accumulators to compute the weight C_{Aj} checksums. At the moment of weight flowing, the accumulators could be busy accumulating the previous matrix multiplication results if needed by the program code, like when tiling is performed. To avoid delaying the pipeline execution or allocating additional hardware resources, we exploit the mapping of the accumulators on the on-chip DSP to operate the accumulators in Single Instruction Multiple Data (SIMD) mode. Indeed, modern FPGA devices support DSP operands on more than 48 bits (AMD Ultrascale 48 bits, AMD Versal 58 bits, Intel Agilex 54 bits). By doing so, while matrix multiplication results on 32 bits are accumulated (baseline behavior), the weights, represented on 8 bits, are accumulated to produce the checksums (testing behavior). Once all the weights for the instruction are read, the computed checksum values are stored in a dedicated location of the registers file, referred to as R0 and R1. The *matrix multiply* instruction always follows the *read weight* instruction, and similarly, *t_matrix_multiply* always follows the *t_read_weight*. In the testing mode, additional test vectors, required to compute the SA checksum C_{SAj} and $\overline{C_{SAj}}$, detailed in the previous section, are appended to the input vectors, generating the checksums. Once the SA checksums are computed, they flow through the accumulators, just like partial products from the SA. However, they are always summed up with R0 and R1 content. This sum follows the operations described by Eq. (4) and Eq. (5) in Section 3. Without faults, the results must be all 0s and all 1s, which are written back to R0 and R1. A XOR-based detection unit compares the contents of R0, R1, C_{SAj} and $\overline{C_{SAj}}$ to detect faults and to perform diagnosis as explained in subsection 3.3. The diagnosis unit notifies any detected error to NEORV using interruption, as well as the detailed information in the status. When the accelerator triggers the interruption, it needs to be reconfigured to fix errors in the datapath.

Considering the DNN workload, a single network layer is typically decomposed in hundreds of matrix multiplications. It is a programmer's choice to execute the full layer in testing mode, hence executing each matrix multiplication with the additional

checksum computation or just some operations during the overall layer executions. When required by NEORV, the TPU starts fetching instructions from its instruction FIFO, prepared by the NEORV processor. In the testing mode, while a new instruction is fetched and executed, a stage of the TPU pipeline will evaluate the testing checksums produced by the previous instruction. If no errors are detected, the pipeline operation will continue normally. Differently, when an error is detected, the TPU triggers an interruption to the RISCV that starts the recovery procedure. In the recovery procedure, the TPU pipeline is flushed first, as no further instructions should be executed while the datapath is compromised. Simultaneously, the NEORV32 acquires detailed information about the specific instruction execution that triggered the error. This fault information is crucial for recovery: after reconfiguration, the program resumes from a correctly executed instruction. This approach ensures that the execution continues from the remaining instructions of the current DNN layer, avoiding a full restart.

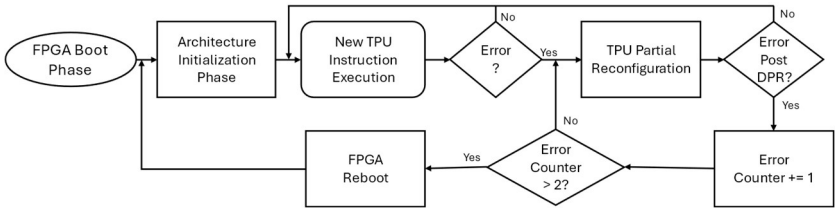


Fig. 3. Error Detection and Correction Flow.

If computations within a layer are fully executed in testing mode, the fault detection latency is confined to a single matrix multiplication operation, allowing all preceding executed instructions to be considered correct. However, if the layer is not processed entirely in testing mode—except for key operations like the first and last matrix multiplications (ensuring correct datapath behavior at entry and exit points of the layer)—the program must restart from the beginning of the faulty layer. This conservative strategy minimizes the potential propagation of undetected faults across layers.

The RISCV processor requests the dynamic partial reconfiguration of the TPU through the DFX controller. Once the partial reconfiguration process is triggered, the NEORV32 waits for its completion. In the proposed design, this is achieved by including in the TPU architecture an *alive* signal that activates as soon as the accelerator becomes operational. The soft processor continuously polls this signal, and as soon as it transitions to `1`, indicating that the TPU is functional again, the interrupt subroutine concludes, and normal execution flow resumes. However, if the partial reconfiguration fails, meaning the soft processor detects consecutive TPU errors during two reconfiguration attempts, the system initiates a full-board reprogramming process to restore the correct behavior. Because the device undergoes complete reconfiguration, the memory content is not retained. This means that the platform performs a cold start. Fig. 3 highlights the flow describe above.

5 Experimental Analysis

5.1 RePAIR implementation details

The RePAIR platform has been implemented on an AMD KCU105 Evaluation Board. The TPU accelerator uses the on-chip DSP blocks to implement all MACs and accumulator units. The weight buffers and unified buffers are mapped to BRAM resources. Minimal LUT usage is required for implementing control logic and glue logic. Table 1 reports the implementation details for the two processing cores.

Table 1. RePAIR Resource Utilization on KCU105.

RePAIR Modules	LUTs	FFs	BRAMs	DSP
tinyTPU	4,294	7,211	181	210
TMR NEORV32	3,219	3180	3	0
DPR Logic	1,185	989	0	0
Glue Logic Resources	13,874	17,670	95.5	3
Total [%]	9.31%	5.99%	46.58%	11.09%

Table 2. Benchmark CNNs Characteristics.

Dataset	Convolutional Layers [#]	Fully Connected layers [#]	Parameters [#]	Accuracy [%]
MNIST	3	1	40,874	97
CIFAR10	6	1	91,648	83.4

The area overhead introduced by interfacing the dynamic partial reconfiguration (DPR) and the two cores, listed as *glue logic* in the table, remain within acceptable limits. All the resources in the design operate at 100MHz. To further increase the reliability of the design, we enabled the ECC in the BRAMs to prevent data corruption due to radiation-induced Single Event Upset (SEU).

5.2 Experiment Analysis

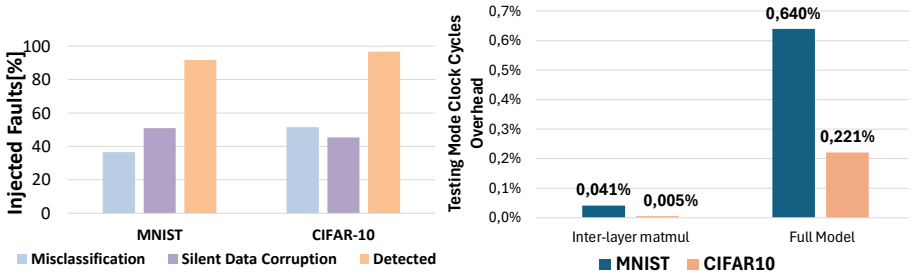
The experimental analysis focuses on two primary aspects of the platform: fault detection capability and dynamic reconfiguration properties. These have been evaluated with experiments using the actual hardware platform. SEU-induced structural faults in the TPU Datapath have been emulated by bitstream corruption to reproduce faults.

To test fault detection capability, a fault injection campaign targeting the resources of the systolic array was conducted. The campaign injected 20,000 distinct faults in the configuration memory. Each fault has been evaluated singularly. Bitflips in the CRAM were used to emulate various fault types, including stuck-at, bridge, conflict, and open faults. As a benchmark, two classification tasks — MNIST digit recognition and CIFAR-10 — served as benchmark applications. The CNNs performing classification were implemented using the quantized Qkeras TensorFlow library and trained from scratch to align with the hardware characteristics of the tinyTPU accelerators,

especially regarding the quantized activation functions integrated within the core. The details of the models' implementation are provided in Table 2. For each injected fault, inference was performed on a random sample of 10 images selected from the test dataset to analyze possible data-induced fault-masking effects during processing. The inferences have been executed in testing mode for all the computations operations.

5.3 Experimental Results and Discussion

The experimental results for fault detection are shown in Fig. 4a. A major concern when evaluating this kind of application is that faults may affect the confidence level of the prediction without changing the results to classification. However, such errors are still relevant since they modify the expected behavior and could eventually lead to misclassification with different inputs. In the chart, we indicated that these errors are Silent Data Corruption (SDC), while the change of the classification output is defined as *misclassification*.



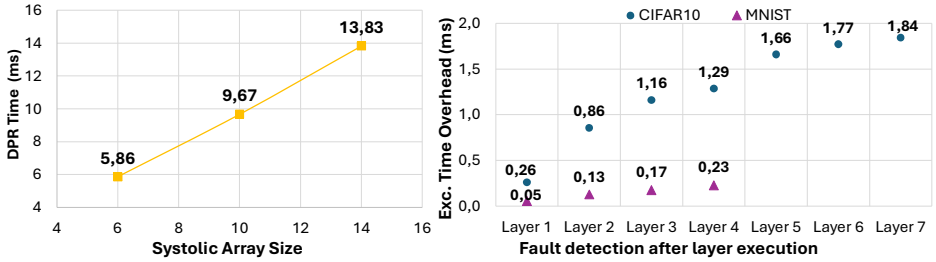
(a) (b)

Fig. 4. (a) Fault Injection Results over 20,000 injected faults and (b) Maximum and Minimum Clock Cycles Overhead

We want to emphasize that the faults that affect designs implemented within the FPGAs due to CRAM corruption are multiple and challenging to detect due to the lack of documentation on CRAM and CDATA, which are not provided by vendors. The proposed approach demonstrated robust fault resilience, achieving an average detection rate of approximately 94%, as shown in Fig 4a. Regarding performance overhead, it is important to highlight that it is related to the inference execution time, which is different for different inference tasks, in particular 0.23ms for MNIST and 1.8ms for CIFAR10, respectively. Additionally, the fault detection mechanism introduces an overhead, but this overhead depends on how many instructions the developer want to execute in testing mode. Reminding that the testing procedure introduces 3 clock cycles of penalty, the execution overhead is *3 clock cycles * number of matmul instructions per layer * number of layers,* in the worst case, i.e. when all the matrix multiplication in all the DNN's layers are executed in testing mode. If the test mode is activated only during the last *matmul* of the layer, we have an overhead with *3 clock cycles * number of layers* while still detecting fault in each layer computation. The minimum and maximum relative overhead for the benchmark models is reported in Fig. 4b. In both cases, the overhead is negligible, and it decreases as the model complexity increases. The disadvantage of having a coarser fault detection is that in case of fault, we need to recompute

the computation of the whole layer instead of repeating only the last matrix multiplication, as will be discussed further. By applying our methodology to the CNN inference, the computational overhead incurred was about 100 clock cycles per layer when executing full testing mode for both architectures, underscoring the platform's efficiency in maintaining fault tolerance and performance.

The advantage regarding system downtime strictly relates to the amount of time required to apply DPR to the accelerator compared to the time required for reconfiguring the full device. The time required for DPR depends on the size of the systolic array since the larger the accelerator, the higher the DPR time, while the time for complete reconfiguration depends on the device size since larger devices require longer reconfiguration times. Specifically, the DPR time resulted in scaling linearly with the systolic array size, as shown in Fig. 5a. For instance, a DPR time of 14ms is necessary to complete the reconfiguration of a 14x14 SA, the maximum SA size allowed by tinyTPU architecture. In contrast, considering the KCU105 device, its full board reconfiguration requires almost 13s. Therefore, when mapping the accelerator to KCU105, we can benefit of an 872.1x reduction in time exploiting DPR. When implementing the platform on different devices, the order of magnitudes remains the same. Indeed, smaller devices have shorter reconfiguration time and fewer resources available, which constrain a smaller SA, reducing the DPR time as well.



(a) (b)

Fig. 5. (a) TPU DPR time related to the SA size ; (b) the execution time overhead in full board reconfiguration, depending on when the fault is detected during the inference.

In the case of full reconfiguration, the program needs to start the inference process from the beginning, assuming it can retrieve the input data again. Consequently, the execution overhead depends on the timing of the fault. In the best-case scenario, the fault is detected during the execution of the first instruction in the first layer of the DNN. Here, the system discards the result after executing the first instruction, reconfigures the board, and then completes the full inference process starting from the beginning. Conversely, in the worst-case scenario, the fault is detected at the end of the inference process, requiring the entire process to be repeated twice. In addition, some time is also required by the NEORV to boot before starting computation. Fig. 5b shows the execution overhead depending on when the fault occurs during the inference, considering the two benchmark CNNs.

When DPR is enabled, the inference process does not start from the beginning after the reconfiguration. When the whole model is executed in testing mode, i.e. every matmul of each layer is executed in testing mode, the process resumes from the last correct

instruction. This approach avoids discarding and executing computations that were correct again. The testing mode ensures the correctness of results up to the fault occurrence. The recovery of the execution from the last correct instructions is achieved by excluding the TPU input/output buffer from the DPR process. The buffer is hardened with ECC, preserving the TPU context during reconfiguration while protecting it from SEU.

The cost of the recovery procedure depends on the cost of the failed instruction, particularly the number of vectors processed by the instruction. Figure 6 illustrates the time overhead associated with executing the faulty instruction as a function of the processed vector count, ranging from a matrix multiplication with the size of operands of 14×14 (matching the TPU dimensions) to operands up to eight times larger. However, the additional time required for processing again the same instruction after DPR, compared to a scenario where no fault is detected, is below 2 µs, which is negligible considering that the entire inference process for the simplest CNN model is 100 times longer.

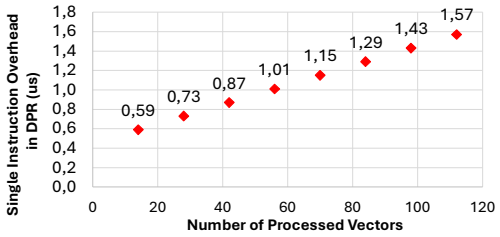
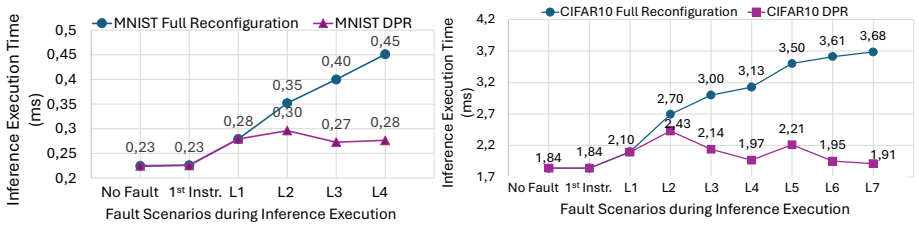


Fig. 6. Faulty instruction overhead in DPR as a function of processed vectors.

However, if the layers are not fully executed in testing mode, for instance, only the first and last instructions are checked to ensure correct inter-layer processing, the program must restart from the last correctly executed layer. Consequently, the total execution time with DPR differs from the previous case and depends on the layers' complexity and the CNN model itself, as explained further.

Figure 7 illustrates a quantitative comparison of inference execution scenarios, focusing on the closest comparable cases involving testing instructions at each layer's entry and exit points, under full reconfiguration versus DPR. In the chart, the No-Fault scenario serves as the baseline, representing the execution with two testing instructions per layer with no fault detected; hence, the inference is executed with no interruptions. The other scenarios depict the total execution time when a fault occurs during different layers, that have different computational costs. These values account only for the computational overhead caused by restarting the program, excluding reconfiguration time. This exclusion is due to the significantly longer duration of full device reconfiguration—on the order of seconds—making it incomparable to the other measured values. The graphs show that the inference execution overhead introduced by partial or full re-execution of the program during inference exhibits similar trends for both CNN benchmarks. If the fault occurs during the execution of the first layer, the computational overhead is the same for both DPR and full reconfiguration. In this case, the only differentiating factors between the two techniques are the total reconfiguration time, which



(a) (b)

Fig. 7. Total inference execution time considering fault occurrences at different stages of the inference process with and without DPR for MNIST (a) and CIFAR10(b).

depends on the device's characteristics (e.g., size, ICAP port speed), and the partial reconfiguration time, which is influenced by the size of the accelerator.

Starting from the second layer, the use of DPR becomes significantly more advantageous. DPR allows the computations completed before the fault to be preserved, enabling the process to resume from the last successfully executed layer. In contrast, with full reconfiguration, the overhead increases as more layers are processed, as all computations must be restarted. For DPR, the worst-case scenario involves re-executing the most complex layer. For the benchmarks under study, this corresponds to the second layer, which results in an increase in inference execution of almost 30% for both MNIST and CIFAR10. In both cases, this is significantly less than the overhead induced by the worst-case scenario in full reconfiguration, which is a more than 96% increase.

Additionally, DPR offers the advantage of significantly shorter reconfiguration times—on the order of milliseconds—compared to full reconfiguration, which can take several seconds for a medium-sized device. These benefits further highlight the efficiency of DPR in mitigating inference overheads during fault recovery.

6 Conclusions

This research introduces RePAIR, a highly reliable and high-performance FPGA-based platform specifically designed for DNN execution in safety-critical applications. RePAIR integrates the NEORV32 RISC-V core, enhanced with TMR for fault tolerance, and tinyTPU, a systolic array-based accelerator augmented with ISA extensions for runtime fault detection during inference. The proposed platform goes beyond fault detection by incorporating fault correction through dynamic partial reconfiguration of the DNN accelerator, managed by the RISC-V core. The platform was tested using hardware fault emulation by corrupting the device configuration memory to simulate radiation-induced single-event upsets affecting the datapath. Experimental results demonstrated a 94% runtime fault detection accuracy across two benchmark CNNs, with a worst-case computational overhead of only 0.6%. Furthermore, employing dynamic partial reconfiguration significantly reduced system downtime, achieving a nearly 900x improvement on medium-to-large-scale FPGA devices.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. “The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2”, Editors Andrew Waterman and Krste Asanovi´c, RISC-V Foundation, May 2017.
2. D. -Z. Li, H. -R. Gong and Y. -C. Chang, "Implementing RISCV System-on-Chip for Acceleration of Convolution Operation and Activation Function Based on FPGA," 2018 14th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT), Qingdao, China, 2018, pp. 1-3, doi: 10.1109/ICSICT.2018.8564810.
3. Héctor Martínez et al, “Parallel GEMM-based convolutions for deep learning on multicore ARM and RISC-V architectures”, Journal of Systems Architecture, vol. 153, 2024, doi: j.sysarc.2024.103186
4. Xingbo Wang et al, “RV-GEMM: Neural Network Inference Acceleration with Near-Memory GEMM Instructions on RISC-V”. in 21st ACM International Conference on Computing Frontiers (CF '24) pp. 302–305. doi: 10.1145/3649153.3649181
5. N. P. Jouppi et al., “In-datacenter performance analysis of a tensor processing unit”, ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), 2017, pp. 1-12
6. F. Libano et al., "Efficient Error Detection for Matrix Multiplication With Systolic Arrays on FPGAs," in *IEEE Transactions on Computers*, vol. 72, no. 8, pp. 2390-2403A
7. S. Burel et al.,"MOZART: Masking Outputs with Zeros for Architectural Robustness and Testing of DNN Accelerators," 2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS), Torino, Italy, 2021, pp. 1-6
8. E. Vacca et al., "ZOR: Zero Overhead Reliability Strategies for AI Accelerators," 2024 22nd IEEE Interregional NEWCAS Conference (NEWCAS), Sherbrooke, QC, Canada, 2024, pp. 248-252, doi: 10.1109/NewCAS58973.2024.10666350.
9. P. E. Dodd, et al., "Current and Future Challenges in Radiation Effects on CMOS Electronics," in *IEEE Transactions on Nuclear Science*, vol. 57, no. 4, pp. 1747-1763, Aug. 2010
10. AMD, Soft Error Mitigation Controller v4.1, (PG036), 2018
11. S. Nolting and Contributors, “The NEORV32 RISC-V Processor.” Zenodo, Aug. 18, 2023. doi: 10.5281/zenodo.8260609
12. Jonas Fuhrmann, “Implementierung einer Tensor Processing Unit mit dem Fokus auf Embedded Systems und das Internet of Things”, Germany, 2018., http://hdl.handle.net/20.500.12738/8527
13. A. Sanchez-Flores, L. Alvarez and B. Alorda-Ladaria, "Accelerators in Embedded Systems for Machine Learning: A RISCV View," *2023 38th Conference on Design of Circuits and Integrated Systems (DCIS)*, Málaga, Spain, 2023, pp. 1-6, doi: 10.1109/DCIS58620.2023.10335969.
14. E. Parisi *et al*., "TitanCFI: Toward Enforcing Control-Flow Integrity in the Root -of-Trust," *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Valencia, Spain, 2024, pp. 1-6, doi: 10.23919/DATE58400.2024.10546873.
15. P. R. Nikiema *et al*., "Towards Dependable RISC-V Cores for Edge Computing Devices," *2023 IEEE 29th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, Crete, Greece, 2023, pp. 1-7, doi: 10.1109/IOLTS59296.2023.10224862.
16. C. Gewehr, L. Luza and F. G. Moraes, "Hardware Acceleration of Crystals-Kyber in Low-Complexity Embedded Systems With RISC-V Instruction Set Extensions," in *IEEE Access*, vol. 12, pp. 94477-94495, 2024, doi: 10.1109/ACCESS.2024.3416812.
17. Z. Wang *et al*., "RTPE: A High Energy Efficiency Inference Processor with RISC-V based Transformation Mechanism," *2024 IEEE 6th International Conference on AI Circuits and*

Systems (AICAS), Abu Dhabi, United Arab Emirates, 2024, pp. 297-301, doi: 10.1109/AICAS59952.2024.10595923.
18. J. Wei, L. Zhang, Z. Yu and D. Liu, "Design Space Exploration for Heterogenous SoC Integrated with Matrix Accelerator," *2020 IEEE 2nd International Conference on Circuits and Systems (ICCS)*, Chengdu, China, 2020, pp. 40-43
19. E. Vacca et al., "Failure rate analysis of radiation tolerant design techniques on SRAM-based FPGAs," Microelectronics Reliability, Volume 138, 2022, doi: j.microrel.2022.114778.
20. S. Azimi et al. "A comparative radiation analysis of reconfigurable memory technologies: FinFET versus bulk CMOS", Microelectronics Reliability, Volume 138, 2022, doi: j.microrel.2022.114733
21. E. M. Aguilar, F. Benevenuti and F. L. Kastensmidt, "Hardening a RISC-V Softcore for Embedded Aerospace Applications in SRAM-based FPGA," 2024 37th SBC/SBMicro/IEEE Symposium on Integrated Circuits and Systems Design (SBCCI), Joao Pessoa, Brazil, 2024, pp. 1-5, doi: 10.1109/SBCCI62366.2024.10703996.
22. Á. B. de Oliveira et al., "Evaluating Soft Core RISC-V Processor in SRAM-Based FPGA Under Radiation Effects," in IEEE Transactions on Nuclear Science, vol. 67, no. 7, pp. 1503-1510, July 2020, doi: 10.1109/TNS.2020.2995729.
23. AMD UltraScale Architecture Soft Error Mitigation Controller LogiCORE IP Product Guide (PG187)
24. M. Safarpour et al., "Algorithm Level Error Detection in Low Voltage Systolic Array" in IEEE Transactions on Circuits and Systems II: Express Briefs, Feb. 2022, vol. 69, no. 2, pp. 569-573.
25. J. Kim, et al., "ZOS: Zero Overhead Scan for Systolic Array-based AI accelerator", 2022 19th International SoC Design Conference (ISOCC), 2022, pp. 360-361.
26. H. Lee, et al., "STRAIT: Self-Test and Self-Recovery for AI Accelerator", in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Sept. 2023, pp. 3092-3104
27. AMD, Vivado Design Suite User Guide: Dynamic Function eXchange (UG909)
28. AMD, Dynamic Function eXchange Controller v1.0 Product Guide (PG374)