Hardware Error Detection with In-Situ Monitoring of Control Flow-Related Specifications

Tomonari Tanaka
Kyoto University
Kyoto, Japan

Takumi Uezono
Hitachi, Ltd.
Yokohama, Japan

Kohei Suenaga
Kyoto University
Kyoto, Japan

Masanori Hashimoto
Kyoto University
Kyoto, Japan

ABSTRACT

In hardware accelerators used in data centers and safety-critical applications, soft errors and resultant silent data corruption significantly compromise reliability, particularly when upsets occur in control-flow operations, leading to severe failures. To address this, we introduce a method for monitoring control flow-related specifications using Petri nets. We validated our method across three designs: convolutional layers in LeNet-5, Gaussian blur in Canny edge detection, and AES encryption. Our fault injection campaign targeting the control registers and primary control inputs demonstrated high error detection rates in both datapath and control logic. Synthesis results show that a maximum detection rate is achieved with a few to around 10 % area overhead in most cases. The proposed detectors quickly detect 88.0% to 99.9% of failures resulting from upsets in internal control registers and perturbation in primary control inputs.

CCS CONCEPTS

• **Hardware → Transient errors and upsets**; **Error detection and error correction**.

KEYWORDS

Soft error, Control flow, Error detection

ACM Reference Format:
Tomonari Tanaka, Takumi Uezono, Kohei Suenaga, and Masanori Hashimoto. 2025. Hardware Error Detection with In-Situ Monitoring of Control Flow-Related Specifications. In *30th Asia and South Pacific Design Automation Conference (ASPDAC '25), January 20–23, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3658617.3697744

1 INTRODUCTION

Hardware accelerators that process tasks like image processing and AI inference are increasingly used in various domains, with heightened demand for reliability in safety-critical applications such as autonomous driving and medical devices [1, 2, 3, 4]. Silent data corruption in data centers accommodating hardware accelerators draws significant attention [5, 6, 7]. Soft errors due to cosmic rays in terrestrial and space environments pose significant reliability concerns for these accelerators across their lifetime [8, 9].

Available methods for evaluating hardware accelerator reliability against soft errors include irradiation experiments and fault injection. The former uses actual radiation to deliver accurate assessments but is limited by time and facility availability. Conversely, fault injection experiments are more flexible, allowing for the controlled injection of bit upsets over time and space, with an option to repeat evaluations as needed. Fault injection targeting hardware accelerators has demonstrated that control registers related to control flow are especially vulnerable to bit upsets [10, 11].

Fault-tolerant methods like instruction redundancy in software and hardware lockstep are proposed to detect soft errors impacting control flow [12, 13, 14]. For hardware, Dual Modular Redundancy (DMR) and Triple Modular Redundancy (TMR) are often used, supplemented by application-specific strategies [15, 16]. However, even TMR has been reported as ineffective against single points of failure, such as shared I/O [17], indicating they cannot cope with input failures. Additionally, existing error detection techniques often target the data and control flows of specific applications [18, 19, 20]. Therefore, establishing a generalized high-coverage error detection method capable of identifying control flow faults, including those that cannot be mitigated by TMR, while ensuring efficient area usage, represents a significant yet crucial challenge.

This work proposes a generic approach for monitoring control flow-related specifications in hardware accelerators for error detection. Our key concept involves constructing Petri nets that represent specifications to monitor control flow. This method allows multiple compact Petri nets to detect most control-flow perturbations caused by both bit-flip and input failure, as well as the resulting incorrect datapath outputs. These compact Petri nets can be integrated into hardware for error detection with minimal hardware overhead, ensuring no false error detection during error-free operations.

2 RELATED WORK

We aim to develop a method for detecting hardware failures efficiently due to soft errors by monitoring hardware control flow. In reviewing relevant research, several strategies emerge:

Hard Error Detection in Datapaths: Periodic monitoring using pre-acquired golden values for implementations based on high-level synthesis (HLS) is reported in [21]. Employing golden data during

ASPDAC '25, January 20–23, 2025, Tokyo, Japan

ACM ISBN 979-8-4007-0635-6/25/01
https://doi.org/10.1145/3658617.3697744

idle times in CNN accelerators allows error detection without affecting cycle counts [18]. However, these methods do not address soft errors directly.

Software Techniques for Transient Error Mitigation: Techniques such as instruction duplication and multi-core redundancy computations have been proposed to counter transient errors [22, 23]. Low-cost error detectors identify vulnerabilities and introduce redundant programming [24], while compiler enhancements automatically duplicate instructions to enhance resilience against soft errors [12]. Although instruction duplication mitigates silent data corruptions from datapath faults, control flow-related faults like hang-ups remain challenging, with TMR showing no improvement.

Fault Injection and Reliability Evaluation: Techniques include focusing fault injections on neuron outputs in CNN accelerators where bit-flips significantly impact results [25] and identifying sensitive bits in binary data in machine learning applications to improve fault injection efficiency [26]. These methods effectively identify vulnerable bits but still face challenges in efficient hardware failure detection.

Soft Error Impact Mitigation: One approach selectively protects crucial registers in control and data flows via HLS to reduce error probability, though it prioritizes error probability reduction over detection [27]. A tool extracts and protects control flow from C-language descriptions for HLS, mainly detecting errors affecting execution times rather than computational accuracy [28]. The performance of TMR techniques has been evaluated through irradiation experiments [17]. In normal TMR configurations, a shared primary input is a single-point of failure, implying that TMR cannot mitigate errors when inputs are affected by soft errors, thus compromising the benefits of redundancy.

Machine Learning for Error Detection: A machine learning-based method has been proposed to monitor crucial signals related to control flow in microprocessors [29]. Despite its general applicability, this method risks misidentifying fault-free operations as faulty, which is a significant problem confirmed in subsequent experiments detailed in Section 5.4.

Monitoring the soundness of control flow and detecting abnormal behaviors efficiently remain substantial challenges. Unlike datapath processes, which are driven by clear algorithms, control flow depends on each design, complicating its efficient monitoring.

3 PROPOSED METHOD

Fig. 1 shows the proposed method consisting of three steps: (1) generating Petri nets from specifications, (2) evaluating their fault detection performance, and (3) selecting Petri nets based on area and fault detection performance and implementing the selected ones as detectors. This work assumes the existence of a specification document that fully describes signal changes within the hardware, which is typical in industrial designs, especially reliability-critical hardware.

3.1 Petri nets

Before explaining each step in detail, we give the definition of Petri nets used in this work. A Petri net is a mathematical model used to describe the behavior of a discrete event system. Structurally, a Petri net S is a directed bipartite graph whose vertices are divided into two sets—*places* P and *transitions* T—connected by directed

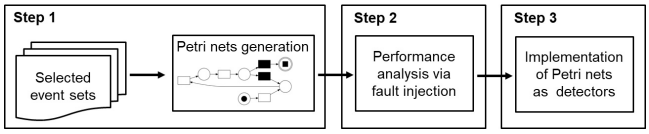

Figure 1: Proposed method from generating Petri nets to implementing those as detectors.

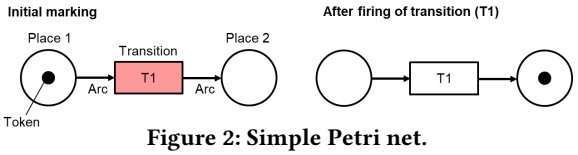

Figure 2: Simple Petri net.

edges E. Each place $p \in P$ keeps a non-negative number of *tokens*. A state of a Petri net is represented by a function $M : P \to \mathbb{N}$, where $M(p)$ denotes the number of tokens in place p. We call such a function f a *marking* of the Petri net. The initial marking of a Petri net is denoted by M_0. A transition $t \in T$ is said to be *enabled* if every place p such that $(p, t) \in E$ (i.e., p is an input place of t) contains at least one token. A Petri-net changes its state by *firing* one of the enabled transitions; once an enabled transition t is fired, it consumes one token from each input place p_I (i.e., $(p_I, t) \in E$) and produces one token to each output place p_O (i.e., $(t, p_O) \in E$). In this way, a Petri net models a sequence of discrete events as a sequence of transition firing.

Fig. 2 shows a simple Petri net with two places and one transition. In the initial marking, Place 1 contains one token, while Place 2 is empty. The transition T1, whose input is Place 1 and output is Place 2, is enabled since Place 1 contains a token. After T1 fires, the token in Place 1 is consumed, and a token is produced in Place 2.

Petri nets have been widely used in studies related to hardware security [30, 31, 32, 33]. However, there has been relatively little research on the efficient implementation of Petri nets as runtime checkers for soft errors. The following sections explain each step, from the construction of Petri nets to their implementation.

3.2 Step 1: Extract event sets from specification and generate their corresponding Petri nets

In the specification document, we categorized specific signal changes into four types, interpreting them as individual events. Types 1 and 2 involve changes in the value of a specific signal; Type 1 includes any change, while Type 2 focuses on changes to specific values. Types 3 and 4 relate to the i-th change of the signal value, where i is a predetermined value. Type 3 captures any i-th changes, whereas Type 4 is restricted to i-th changes of specific values. For example, considering a status signal indicating two states (*S1*, *S2*), changing the signal value represents a state transition. A simple state transition is Type 1, while a transition to a specific state (e.g., *S2*) is Type 2. The i-th state transition is Type 3, and the i-th transition to a specific state (e.g., i-th *S1*) is Type 4. Considering the implementation, the allocation type of an event is relevant to the hardware resources required for its observation. For instance, observing a Type 4 event requires more hardware resources than observing a Type 1 event to count the number of transitions. We finally consider the balance between hardware resources and error detection performance, which will appear in Section 5.3.

Given our focus on monitoring the control flow of hardware accelerators, we extract an event set that meets a specific condition: *There must be at least two target events, and their occurrence order must remain consistent across multiple executions.* For instance, this condition is met if a monitored specification includes events A, B, and C; and if these events consistently occur in the order of A, B, and then C during correct executions. In this paper, we manually identify event sets. Meanwhile, large language models (LLMs) or automated assertion techniques, e.g., [34, 35, 36], may help.

Next, we generate Petri nets corresponding to individual event sets. Each event is assigned to a transition, and a Petri net is constructed to represent the sequence of these event occurrences. To enhance monitoring capabilities, multiple event sets and their Petri nets are generated. We use Petri nets to handle complex control flows in anticipation of future demands. However, within the scope of this paper, alternatives such as automata may also be applied.

3.3 Step 2: Evaluate error detection performance

3.3.1 Simulating Petri net-based error detection. Error detection with Petri nets is achieved by monitoring the sequence of transition firings. If an abnormal firing sequence (including the firing of an incorrect final transition) that is not defined by the Petri net is detected, it is considered that an error has been detected. The focus here is on monitoring control flow; thus, fault injections simulate bit flips in registers responsible for control flow, as well as in primary control inputs. Detection of output errors is indicated by abnormal transition firings, where the output error is a fault resulting in incorrect outcomes. The output errors are categorized either as incorrect computational results, namely silent data corruption defined by deviations from correct values, or as abnormal terminations of processing, identified by timing. These faults can affect both the datapath and control logic. When a Petri net detects an error upon the occurrence of an output error, it is considered true error detection.

3.3.2 Metrics of error detection performance. We evaluate the error detection performance of Petri nets using three metrics: error detection rate, error detection latency, and false error detection for error-free operations. When an error is detected in the Petri net, it is expected to indicate that the hardware produces incorrect outputs. Therefore, if the Petri net detects an error and the hardware outputs a fault, it is counted as a true positive error detection (N_{TP}). Then, the error detection rate (*DR*) is defined as $DR = N_{TP}/N_{OE}$, where N_{OE} is the number of output error occurrences.

Additionally, we evaluate the latency of error detection using Petri nets. Latency (*Lat*) is defined as the average number of clock cycles between the injection of a fault and the Petri net detecting the error. In scenarios where the process fails to complete and results in a timeout, the Petri net may only detect the incorrect final transition. Given that the practical timeout duration is not fixed, the latency of error detection becomes ambiguous. Consequently, the proportion of such error detections is calculated as *DR_TO*, which is a subset of the *DR*. A smaller *Lat* and a lower *DR_TO* are indicative of better error detection capabilities.

A critical measure of the effectiveness of Petri net-based error detection is its ability to avoid false positives during error-free (golden) operations. We introduce a metric, *FP_golden*, which can be Yes or No, to assess the presence of false error detection. In the proposed scheme, *FP_golden* is guaranteed to be *No*, indicating there are no false positives. This assurance stems from the Petri nets being generated directly from the specification documents. The absence of false error detection will be further validated when our method is applied to the three designs discussed later.

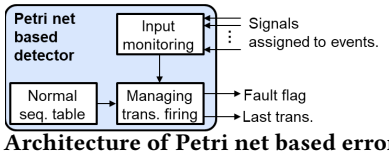

Figure 3: Architecture of Petri net based error detector.

3.4 Step 3: Select and implement Petri nets as error detectors

This step involves implementing Petri nets as error detectors to monitor hardware failures in real time. Fig. 3 illustrates the architecture of the Petri net-based error detector, consisting primarily of three parts: input monitoring, managing transition firing, and the normal sequence table. The input to the error detector consists of the signal lines assigned to events. These input signals are monitored by the input monitoring module to detect changes in signals and the associated events. Event occurrences are defined as the transition firing in the Petri nets. The transition-firing management module monitors the firing sequence of transitions based on the normal sequence table. When the module detects transitions firing in abnormal sequences, it asserts the fault flag (Fault flag) to indicate error detection. Additionally, the transition-firing management module constantly outputs the last-fired transition (Last trans.), enabling the detection of abnormal process terminations.

When maximizing detection rate (DR) with an area overhead constraint, we find the combination of detectors that achieves the highest DR while satisfying the area constraint. When minimizing area with a DR constraint, we identify the detector combination with the minimum area overhead while meeting the DR constraint.

4 DESIGN EXAMPLES

4.1 Convolutional layer computation in CNN

For practical applications, such as those used in autonomous driving, we first focus on the convolutional layer computation in LeNet-5 on a CNN accelerator, specifically a quantized LeNet-5 model, employing INT8 precision and trained on the MNIST dataset. The convolutional layer computation under test has $32 \times 32 \times 1$ input activation data and produces a $28 \times 28 \times 6$ output. Post-convolution, rectified linear unit (ReLU) activation functions are applied.

Fig. 4 illustrates the architecture of the CNN accelerator proposed in [37], which enables high data reuse and low latency performance. The convolution (Conv.) has primary control inputs from the control module. The input data, including weight data and activation data, is fed into the Weight buffer and Activation buffer, respectively. The WT FSM and Data FSM modules manage the input data from the buffers and control the main computation on the PE Array. The Delay CTR and MAC CTR also manage computation on the PE Array. The processed data in the PE Array is accumulated by the Accumulation module via the Temporal buffer and outputs the computation results to the Direct Memory Access module via the Output module.

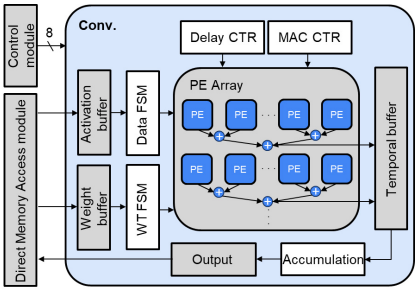

Figure 4: Architecture of CNN accelerator. White boxes are main processing modules.

To construct Petri nets that monitor the control flow, we first organize event sets from the specifications. Table 1 details the event sets, their corresponding IDs for Petri nets, and the event assignment types used. Fig. 5 displays 14 generated Petri nets, each corresponding to an event set. Transition labels correspond to the event labels in Table 1. For example, the Petri net for CONV_3 includes three defined events. This net features a path initiating with the first transition (12) and includes recurring transitions (13, 14). Black transitions indicate branches, with the bottom transition firing at the final loop, signifying the completion as the token moves to the rightmost place.

Logic synthesis for the CNN accelerator, along with 14 Petri nets, was conducted on the Kintex UltraScale FPGA with part xcku035-fbva900-1-i using the Vivado tool. The CNN accelerator used 20,936 LUTs and 17,739 FFs, and the 14 Petri nets used 2,050 LUTs and 1,045 FFs. These 14 Petri nets are just candidates, and a part of them will be selected as detectors. Additionally, we investigated the impact on the maximum operating frequency. While the maximum operating frequency in the original design was 119 MHz, it dropped to 118 MHz when the error detectors were added. The speed impact of Petri net-based error detectors was negligible.

4.2 Gaussian blur in canny edge detection

To examine applicability of our method in general image processing, we focus on Gaussian blur in Canny edge detection. The architecture of Gaussian blur (Gaus.) is illustrated in Fig. 6. Gaus. processes the input image data according to the AXI4-Stream protocol. AXI4-Stream consists of a signal for data transmission (data) and control related signals (user, valid, last, ready). Gaus. receives input images with dimensions of 64×48. The input of AXI4-Stream is acquired by the receiving module (AXIS recv.), then processed by the main processing module (Gaussian blur calc.) via First-In, First-Out (FIFOs). Subsequently, the module (AXIS send) outputs the calculation result according to the AXI4-stream protocol. This process executes in a pipeline with per-clock-per-pixel-processing [38].

Table 2 indicates the monitored event sets, their associated IDs, and the used event assignment types. Three Petri nets have been generated, each corresponding to a ID. The three modules operate synchronously for pipeline processing. By including signals from each module within the event set, the synchronous operation of the three modules is monitored.

In logic synthesis, 581 LUTs and 577 FFs for Gaus., and 493 LUTs and 329 FFs for 3 Petri nets are utilized in the Zynq-7000 FPGA with part xc7z020clg484-1. The FPGA differs in convolutional layer

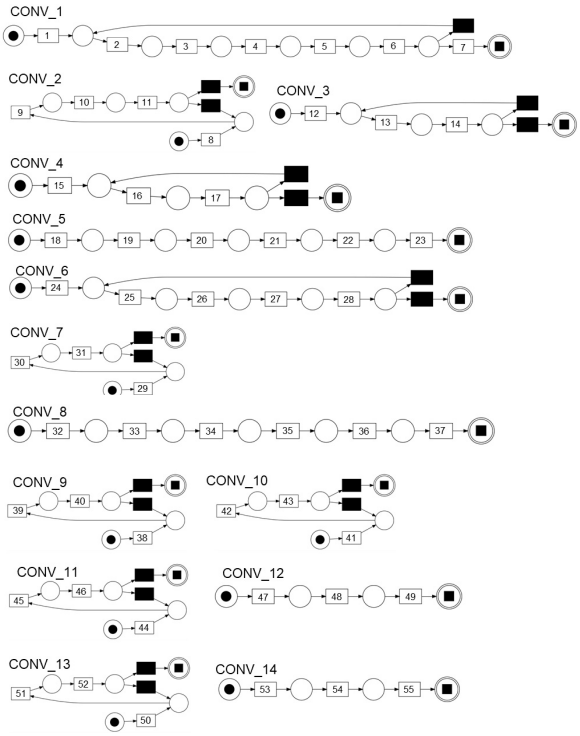

Figure 5: Petri nets for monitoring control-flow in Conv.

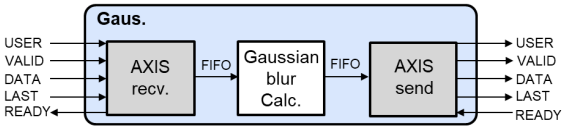

Figure 6: Architecture of Gaussian blur. A white box is the main processing module.

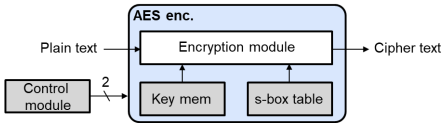

Figure 7: Architecture of AES encryption. A white box is the main processing module.

computation and Gaussian blur, it essentially does not affect the error detection performance of Petri nets.

4.3 Advanced encryption standard (AES)

As a distinct example from the aforementioned two image processing techniques, we target Advanced Encryption Standard (AES) encryption, which requires high reliability. The architecture of the AES encryption system implemented is depicted in Fig. 7, as described in [39]. Our focus was to construct a Petri net that targets the sequential encryption of five 128-bit plaintexts.

Table 3 details the monitored event sets. Using the Vivado tool, the AES encryption system and its seven associated Petri nets were synthesized, targeting the Zynq-7000 FPGA. 2,525 LUTs and 2,331 FFs for AES enc., and 281 LUTs and 205 FFs for 7 Petri nets are utilized.

Table 1: Monitored event sets, their corresponding IDs, and used event assignment types in Conv.

ID	Event set with Event Label (#)	Type
CONV_1	Initiation of processing (1), updating of horizontal counter for activation data (2), retrieval of data cube (3), completion of data cube retrieval (4), writing to FIFO buffer (5), completion of convolution calculation for the data cube (6), conclusion of all computations (7).	2
CONV_2	Initiation of processing (8), permission for data cube computation (9), initiation of data cube computation (10), updating of data cube (11).	2
CONV_3	Initiation of processing (12), retrieval of activation data from a specific position (13), updating of vertical counter for activation data (14).	2,3
CONV_4	Initiation of processing (15), permission for weight data retrieval (16), retrieval of weight data (17).	2,3
CONV_5	Setting of input channel number (18), setting of output channel number (19), initiation of processing (20), state change for processing (21), computation of a specific data cube (22), state change for completion (23).	2,3
CONV_6	Initiation of processing (24), retrieval of weight data corresponding to the activation data cube (25), updating of a coordinate (26), completion of data cube computation (27), permission for next computation (28).	2
CONV_7	Initiation of processing (29), retrieval of specific weight data (30), verification of specific weight data retrieval (31).	2,3
CONV_8	Setting of output channel number to WT FSM (32), initiation of processing (33), state change for weight data acquisition (34), acquisition of the final weight data (35), completion of weight data acquisition (36), conclusion of all computations (37).	2,3
CONV_9	Initiation of processing (38), output of specific data (39), completion of specific data output (40).	2,3
CONV_10	Initiation of processing (41), permission for processing from Delay MAC. (42), initiation of primary output (43).	2,3
CONV_11	Initiation of processing (44), output of specific data from Delay MAC (45), primary output of specific data (46).	2,3
CONV_12	Retrieval of the final activation data (47), output of specific data (48), completion of specific data output (49).	2,4
CONV_13	Initiation of processing (50), retrieval of specific data cube (51), updating of weight data (52).	2,3
CONV_14	Initiation of processing (53), completion of weight data retrieval (54), updating of specific address for activation data (55).	2,3

Table 2: Monitored event sets in Gaussian blur.

ID	Event set with Event Label (#)	Type
GAUS_1	Targeting per-line processing, initiation in AXIS recv. (1), initiation in Gaussian blur calc. (2), initiation in AXIS send (3), completion in AXIS send (4).	2
GAUS_2	Targeting vertical counters, update in AXIS recv. (5), update in Gaussian blur calc. (6), update in AXIS send (7). Acquisition of specific pixel data in AXIS recv. (8).	1,3
GAUS_3	Targeting specific pixel data, acquisition in AXIS recv. (9), writing to FIFO in AXIS recv. (10), acquisition in Gaussian blur calc. (11), writing to FIFO in Gaussian blur calc. (12), acquisition in AXIS send (13), output in AXIS send (14), completion of per-line processing in AXIS send (15). Completion of image processing (16).	2,3

Table 3: Monitored event sets in AES encryption.

ID	Event set with Event Label (#)	Type
AES_1	Initiation of processing (1), permission for state change (2), permission for round update (3), update of round (4).	2
AES_2	Initiation of processing (5), permission for state change (6), reset of per-round S-box (7).	2,3
AES_3	Initiation of processing (8), permission for state change (9), acquisition of per-round plaintext (10).	2,3
AES_4	Initiation of processing (11), start of processing per-plaintext (12), permission for next plaintext (13), completion per-plaintext (14).	2,3
AES_5	Initiation of processing (15), increment of per-round S-box counter (16), acquisition of S-box for next round (17).	2,3
AES_6	Initiation of processing (18), permission to update round counter (19), update of round counter (20).	2,3
AES_7	Permission for state change (21), state change for processing (22).	2,3

5 EXPERIMENTAL RESULTS

5.1 Experimental setup

To assess the efficacy of our error detectors, we perform RTL fault injection simulations on the three target designs in two cases. In Case 1, faults are injected into control registers within the target design, assuming a direct impact of soft errors on the target design. For experimental efficiency, we limited fault injections to the main processing module of each design.

In Case 2, faults are injected into the primary control inputs of the target design, assuming that faults are propagating from upstream circuits. We intentionally randomized primary control inputs across ten consecutive cycles. To justify this fault injection approach, we conducted preliminary experiments involving over 600 instances of bit-flip fault injections on parts of the control modules of Conv. and AES enc., resulting in erroneous primary control inputs appearing for more than 14,512 and 15 cycles on average, respectively. Similarly, the bit-flip in Gaus. exhibited prolonged incorrect outputs on AXI4 streams, implying that primary control inputs can receive similar faults. These results indicate that our fault injection setup in Case 2 is not excessive but practical.

5.2 Baseline error detection performance

For Case 1 fault injection, Table 4 shows the error detection performance on the three designs, where all Petri nets are utilized. $N_{regs}(N_{bits})$ indicates the number of control registers and the total

Table 4: Error detection performance in Case 1: detection rate *DR*, error detection latency *Lat*, and false error detection for error-free golden operations *FP_golden*.

Design	N_{regs} (N_{bits})	$N_{inj.}$	N_{OE}	*DR* (%)	*Lat*(cycles)	*FP_golden*
Conv.	29 (246)	43,500	31,898	99.5	107.6	*No*
Gaus.	11 (35)	72,600	53,638	88.0	53.4	*No*
AES enc.	4 (9)	40,000	26,240	95.3	3.8	*No*

Table 5: Error detection performance in Case 2.

Design	N_{input}	$N_{inj.}$	N_{OE}	*DR* (%)	*Lat*(cycles)
Conv.	8	10,000	9,911	99.9	2.5
Gaus.	4	10,000	9,668	96.3	102.1
AES enc.	2	40,000	18,310	99.9	1.0

number of bits they include, which are targeted for fault injection. $N_{inj.}$ indicates the total number of fault injections performed. An equal number of fault injections were conducted for each target register within a design. N_{OE} represents the occurrence count of output errors. Here, the output errors include both incorrect computation results and abnormal termination of processing. In N_{OE}, 87.6% corresponds to incorrect calculation results in Conv., 68.6% in Gaus., and 81.4% in AES enc., respectively. The remaining percentages in each design correspond to abnormal termination of processing. *DR* signifies the error detection rate.

When considering *DR*, Conv. achieved the highest detection rate of 99.5%, surpassing the other designs due to the larger number of Petri nets. In contrast, Gaus. exhibited the lowest error detection rate *DR* of 88.0%. The number of Petri nets is three, and their diversity might be limited. Considering that the clock cycles for normal processing are 20,521 cycles for Conv., 8,676 cycles for Gaus., and 432 cycles for AES enc., the *Lat* is small for all designs, indicating that the fast error detection is achieved. The *DR_TO* was 0.1% for Conv., 0.5% for Gaus., and 12% for AES encryption, indicating particularly excellent for Conv. and Gaus.. The construction method of Petri nets guarantees *FP_golden* to be *No* in every design, ensuring there are no alarms of error detection during golden error-free operations.

Table 5 shows the error detection performance in Case 2. N_{input} indicates the number of primary control inputs. In N_{OE}, 100.0% corresponds to incorrect calculation results in Conv., 98.6% in Gaus., and 92.6% in AES enc., respectively, while the remaining percentages indicate abnormal termination of processing. The *DR* achieved over 96% in all three designs. Redundancy techniques such as TMR cannot mitigate faults in primary inputs if each module receives the same faulty inputs because all modules misbehave identically. Meanwhile, the proposed error detector detects control-flow disturbances from faulty inputs and the resulting incorrect outputs. Along with short *Lat*, the *DR_TO* was 0.0% for Conv., 0.3% for Gaus., and 0.0% for AES encryption. This result indicates fast error detection.

Although Petri net detectors themselves may be affected by soft errors, they do not interfere with the monitored circuit because a Petri net detector does not have an output for the monitored circuit. We empirically confirmed this via bit-flip fault injection in one Petri net (AES_1). The result shows that faults in the Petri net detector never affect the monitored circuit.

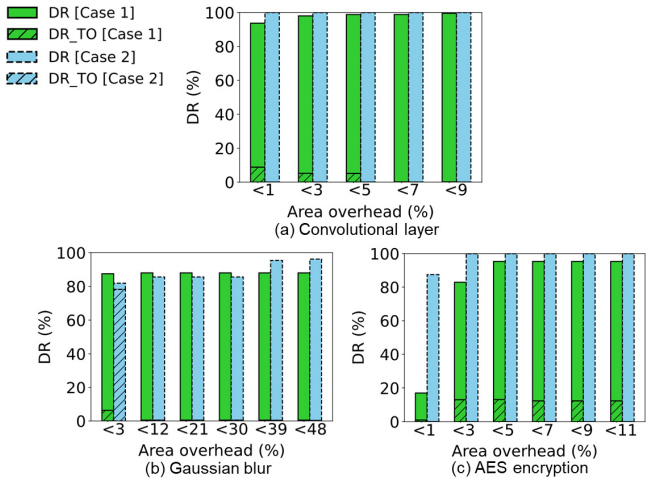

Figure 8: Trade-off between area overhead and DR.

5.3 Trade-off between area and DR

Fig. 8(a) depicts the relationship between area overhead and error detection rate *DR*, concerning convolutional layer computation. The area overhead is calculated based on the number of LUTs. The x-axis represents the thresholds for area overhead. The left-side bar illustrates the maximum *DR* at each threshold in Case 1, along with the subset *DR_TO*. The right-side bar similarly illustrates *DR* and *DR_TO* in Case 2. The maximum *DR* at each threshold is calculated by considering all combinations of Petri nets that satisfy that threshold. When all 14 Petri nets are utilized, the area overhead is approximately 9%. In Case 1, the *DR* gradually increases until reaching this 9%, while decreasing the *DR_TO*. This indicates efficient improvement. Meanwhile, a mere 1% of area cost achieves the *DR* of 93.7% in Case 1, and 99.9% in Case 2. The *DR_TO*, consistently at 0% in Case 2, indicates that an error was detected rapidly.

Similarly, the relationship between the area overhead of Petri nets and *DR* in Gaussian blur is shown in Fig. 8(b). For Case 1, The maximum *DR* is reached by an area overhead of up to 12%, and it is observed that the *DR* does not increase with area overhead exceeding 12%. In both Case 1 and Case 2, the *DR_TO* decreases significantly with over 12% area overhead, indicating a significant improvement in error detection latency. In Case 2, the *DR* exceeds 80% already at <3%, gradually improving.

Fig.8(c) shows the result of AES encryption. The area overhead when using all seven Petri nets is approximately 10%. However, in Case 1 the maximum *DR* is reached when the area overhead is <5%, with *DR_TO* approximately at 13%. Conversely, it demonstrates a sharp increase in *DR* with each 1% increase in area overhead until <3%. In Case 2, the maximum *DR* reached 99.9% at <3%, with *DR_TO* consistently at 0%, indicating the rapid error detection.

These results demonstrate that the proposed method allows for effective consideration of adding or removing Petri nets based on the trade-off between area overhead and error detection rate. This enables flexible adaptation to circuit area constraints and high error detection rate requirements.

5.4 Comparison with related work

To comparatively evaluate the proposed method, we implemented a machine learning-based error detector [29], as a baseline. The features in the dataset focus on the values of selected signals. For

Table 6: Results of machine learning-based error detectors in Case 1 fault injection.

Design	$N_{inj.}$	N_{OE}	*DR* (%)	*Lat* (cycles)	*FP_golden*
Conv.	14,500	10,595	99.8	4152.4	***Yes***
Gaus.	11,000	8,108	91.7	284.8	*No*
AES enc.	36,000	23,630	58.4	0.7	*No*

the convolutional layer computation, we selected 29 registers, 11 registers for Gaussian blur, and 4 registers for AES encryption as elements of the features. These selected registers are those targeted for fault injection in Section 5.2. We set the signal value history over five clock cycles as each individual feature, which is determined to be the most effective parameter reported in [29]. To prepare a training dataset, we conducted 600 RTL simulations for convolutional layer computation, 1,200 for Gaussian blur, and 2,000 for AES encryption. These RTL simulations consist of an equal distribution of golden and fault injection data, with a ratio of 1:1. We employed the XGBoost [40] machine learning algorithm to train a model, which serves as the machine learning-based error detector. We set the hyperparameters as follows: learning rate to 0.3, maximum tree depth to 6, and the number of decision trees to 10. Fault injection was performed on the same registers as those in Case 1 in Section 5.2.

Table 6 shows the performance of the machine learning-based error detector. Within N_{OE}, 87.6% corresponds to incorrect calculation for Conv., 68.2% for Gaus., and 81.4% for AES enc., with the remainder corresponding to abnormal termination of processing. The error detection rate for convolutional layer and Gaussian blur is slightly higher than that of our method. In contrast, when targeting AES encryption, the error detection rate is below 60%, indicating low error detection performance even though *Lat* (latency) is very short. This is because there are only four registers observed in this circuit, making it difficult for the machine-learning-based technique to capture the characteristic signal changes in the fault cases.

However, false errors were detected in the convolutional layer even during the execution of golden simulations. Therefore, *FP_golden* is *Yes*. In practical scenarios, given the low probability of soft errors, it is generally assumed that most instances operate error-free. If *FP_golden* is *Yes*, it implies that an error is detected with each 2 ms execution of the convolutional layer operation, which is practically unacceptable. On the other hand, with the proposed Petri net-based error detector, errors are not detected when faults are not injected.

We remark that our goal is a hardware implementation of soft-error detectors, which is demonstrated to be possible in our approach, whereas it is not clear how their machine-learning–based approach can be implemented using hardware [29]. Implementing an XGboost model is expected to be more challenging, but it is one of our future work.

6 CONCLUSION

This paper presented an error detection method based on monitoring control flow-related specifications in hardware accelerators using Petri nets. We successfully developed a comprehensive methodology for implementing Petri net-based detectors from design specifications. Our methodology was validated through fault injection tests on the convolutional layers of LeNet-5, achieving a detection rate of 99.5% for register bit-flips with an area overhead limited to 9%, and detection rate of 99.9% for primary input. This approach enables comprehensive detection of errors caused by both bit-flip and input failure while minimizing area overhead. A notable advantage of our method is the complete absence of false error detections in error-free operations, making it highly suitable for practical applications.

ACKNOWLEDGMENTS

This study was partially supported by JSPS KAKENHI Grant Number 24H00073, and JST SPRING, Grant Number JPMJSP2110.

REFERENCES

[1] Paolo Rech. 2024. Artificial Neural Networks for Space and Safety-Critical Applications: Reliability Issues and Potential Solutions. *IEEE Transactions on Nuclear Science*, 71, 4, 377–404.
[2] S. Jha, et al. 2019. ML-Based Fault Injection for Autonomous Vehicles: A Case for Bayesian Fault Injection. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. (June 2019), 112–124.
[3] F. Fausti, et al. 2019. Single Event Upset tests and failure rate estimation for a front-end ASIC adopted in high-flux-particle therapy applications. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 918, 54–59.
[4] T. Tanaka, et al. 2022. Impact of Neutron-Induced SEU in FPGA CRAM on Image-Based Lane Tracking for Autonomous Driving: From Bit Upset to SEFI and Erroneous Behavior. *IEEE Transactions on Nuclear Science*, 69, 1, 35–42.
[5] B. Bittel, et al. 2024. Data Center Silent Data Errors: Implications to Artificial Intelligence Workloads & Mitigations. In *2024 IEEE International Reliability Physics Symposium (IRPS)*, 1–5.
[6] Andrew M. Keller, et al. 2021. The Impact of Terrestrial Radiation on FPGAs in Data Centers. *ACM Trans. Reconfigurable Technol. Syst.*, 15, 2, Article 12, (Dec. 2021), 21 pages.
[7] S. Konno, et al. 2024. Exploration of Fault Identification and Automatic Recovery in Cloud-based FPGA Systems. In *2024 IEEE International Conference on Consumer Electronics (ICCE)*, 1–6.
[8] Israel C. Lopes, et al. 2018. Reliability analysis on case-study traffic sign convolutional neural network on APSoC. In *2018 IEEE 19th Latin-American Test Symposium (LATS)*, 1–6.
[9] B. Du, et al. 2019. Ultrahigh Energy Heavy Ion Test Beam on Xilinx Kintex-7 SRAM-Based FPGA. *IEEE Transactions on Nuclear Science*, 66, 7, 1813–1819.
[10] J. Hoefer, et al. 2023. SiFI-AI: A Fast and Flexible RTL Fault Simulation Framework Tailored for AI Models and Accelerators. In *Proceedings of the Great Lakes Symposium on VLSI 2023* (GLSVLSI '23). Association for Computing Machinery, Knoxville, TN, USA, 287–292.
[11] S. Sabogal, et al. 2021. Reconfigurable Framework for Resilient Semantic Segmentation for Space Applications. *ACM Trans. Reconfigurable Technol. Syst.*, 14, 4, Article 22, (Sept. 2021), 32 pages.
[12] M. Bohman, et al. 2019. Microcontroller Compiler-Assisted Software Fault Tolerance. *IEEE Transactions on Nuclear Science*, 66, 1, 223–232.
[13] X. Iturbe, et al. 2016. A Triple Core Lock-Step (TCLS) ARM® Cortex®-R5 Processor for Safety-Critical and Ultra-Reliable Applications. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, 246–249.
[14] Ádria Barros de Oliveira, et al. 2018. Lockstep Dual-Core ARM A9: Implementation and Resilience Analysis Under Heavy Ion-Induced Soft Errors. *IEEE Transactions on Nuclear Science*, 65, 8, 1783–1790.
[15] F. Libano, et al. 2019. Selective Hardening for Neural Networks in FPGAs. *IEEE Transactions on Nuclear Science*, 66, 1, 216–222.
[16] Timoteo G. Bertoa, et al. 2023. Fault-Tolerant Neural Network Accelerators With Selective TMR. *IEEE Design & Test*, 40, 2, 67–74.
[17] Matthew J. Cannon, et al. 2020. Improving the Reliability of TMR With Non-triplicated I/O on SRAM FPGAs. *IEEE Transactions on Nuclear Science*, 67, 1, 312–320.
[18] W. Li, et al. 2020. Soft Error Mitigation for Deep Convolution Neural Network on FPGA Accelerators. In *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 1–5.
[19] Siva Kumar Sastry Hari, et al. 2022. Making Convolutions Resilient Via Algorithm-Based Error Detection Techniques. *IEEE Transactions on Dependable and Secure Computing*, 19, 4, 2546–2558.
[20] Y. Ibrahim, et al. 2020. Soft errors in DNN accelerators: A comprehensive review. *Microelectronics Reliability*, 115, 113969.

[21] Z. Zhu, et al. 2020. Light-Weight Soft-Errors Detection Mechanism in High-Level Synthesis. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 1–5.

[22] M. Didehban, et al. 2024. Generic Soft Error Data and Control Flow Error Detection by Instruction Duplication. *IEEE Transactions on Dependable and Secure Computing*, 21, 1, 78–92.

[23] M. Didehban, et al. 2016. nZDC: A compiler technique for near Zero Silent Data Corruption. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 1–6.

[24] Siva Kumar Sastry Hari, et al. 2012. Low-cost program-level detectors for reducing silent data corruptions. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, 1–12.

[25] Mohammad Hasan Ahmadilivani, et al. 2023. DeepVigor: VulnerabIlity Value RanGes and FactORs for DNNs' Reliability Assessment. In *2023 IEEE European Test Symposium (ETS)*, 1–6.

[26] Z. Chen, et al. 2019. BinFI: an efficient fault injector for safety-critical machine learning systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (SC '19) Article 69. Association for Computing Machinery, Denver, Colorado, 23 pages.

[27] L. Chen, et al. 2014. Reliability-aware register binding for control-flow intensive designs. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 1–6.

[28] Shane T. Fleming, et al. 2016. StitchUp: Automatic control flow protection for high level synthesis circuits. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 1–6.

[29] N. Nosrati, et al. 2022. MLC: A Machine Learning Based Checker For Soft Error Detection In Embedded Processors. In *2022 IEEE 28th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 1–5.

[30] B. Guechi, et al. 2023. Hardware security module cryptosystem using petri net. *Indonesian Journal of Electrical Engineering and Informatics (IJEEI)*, 11, (June 2023).

[31] L. Patzina, et al. 2010. Monitor petri nets for security monitoring. In *Proceedings of the International Workshop on Security and Dependability for Resource Constrained Embedded Systems* Article 3. Association for Computing Machinery, Vienna, Austria, 6 pages.

[32] S. Bai, et al. 2021. An improved petri net for fault analysis of an electronic system with hybrid fault of software and hardware. *Engineering Failure Analysis*, 120, 105077.

[33] P. Wang, et al. 2007. Fault tolerance of multiprocessor-structured control system by hardware and software reconfiguration. In *2007 International Conference on Mechatronics and Automation*, 3745–3749.

[34] T. Zhang, et al. 2017. Automatic Assertion Generation for Simulation, Formal Verification and Emulation. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 471–476.

[35] S. Germiniani, et al. 2022. HARM: A Hint-Based Assertion Miner. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41, 11, 4277–4288.

[36] W. Fang, et al. 2024. AssertLLM: Generating and evaluating hardware verification assertions from design specifications via multi-LLMs. *arXiv preprint arXiv:2402.00386*.

[37] M. Huang, et al. 2022. A High Performance Multi-Bit-Width Booth Vector Systolic Accelerator for NAS Optimized Deep Learning Neural Networks. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 1–13.

[38] A. Yamawaki, et al. 2018. A Describing Method of An Image Processing Software in C for A High-level Synthesis Considering A Function Chaining. *IEICE Transactions on Information and Systems*, E101D, 2, (Feb. 2018), 324–334.

[39] B. Degnan. 2021. Verilog Implementation of the Symmetric Block Cipher AES (NIST FIPS 197). https://github.com/secworks/aes. (2021).

[40] T. Chen, et al. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (KDD '16). Association for Computing Machinery, San Francisco, California, USA, 785–794.