Graph Learning-based Fault Criticality Analysis for Enhancing Functional Safety of E/E Systems

Sanjay Das[1*], Shamik Kundu[1], Pooja Madhusoodhanan[2], Prasanth Viswanathan Pillai[2], Rubin Parekhji[2], Arnab Raha[3], Suvadeep Banerjee[3], Suriyaprakash Natarajan[3] and Kanad Basu[1]

[1] *University of Texas at Dallas, TX, USA*; [2] *Texas Instruments, Bangalore, India*; [3] *Intel Corporation, CA, USA*

Abstract

The increasing complexity of Electrical and Electronic (E/E) systems underscores the need for protective measures to ensure functional safety (FuSa) in high-assurance environments. This entails the identification and fortification of vulnerable nodes to enhance system reliability during mission-critical scenarios. Traditionally, the assessment of E/E system reliability has relied on fault injection (FI) techniques and simulations. However, FI faces challenges in coping with escalating design complexity, including resource demands and timing overheads. Furthermore, it falls short in identifying critical components that may lead to functional failures. To address these challenges, we propose a Machine Learning (ML)-based framework for predicting critical nodes in hardware designs. The process begins with constructing a graph from the design netlist, forming the foundation for training a Graph Convolutional Network (GCN). The GCN model utilizes graph node attributes, node labels, and edge connections to learn and predict critical nodes in the circuit. The model furnishes up to 93.7% accuracy in identifying vulnerable circuit nodes during evaluation on diverse designs such as Synchronous Dynamic Random Access Memory (SDRAM) controller, OpenRISC 1200 (OR1200) modules. Furthermore, we incorporate an explainability analysis to interpret individual node predictions. This analysis discerns the critical design factors influencing fault criticality in the design. Moreover, to the best of our knowledge, we, for the first time, perform a regression analysis to generate node criticality scores, quantifying the degrees of criticality, that can enable prioritizing resources towards critical nodes.

Keywords: Fault injection analysis, Graph Convolutional Networks

1 Introduction

With the growing design complexity, reducing feature sizes, and time-to-market, hardware systems are increasingly exposed to a diverse spectrum of potential failures. Such failures primarily engender from faults manifested in the hardware. This necessitated the emergence of Functional Safety (FuSa) standards, such as ISO 26262 as a technical framework for the detection and management of faults, with the primary goal of achieving and maintaining a safe operational state [2, 6]. Faults in a critical circuit have the potential to cause catastrophic repercussions during in-field operation [8][11][4]. For instance, a fault manifested in the control circuit of an autonomous vehicle can cause it to accelerate beyond control, leading to potential accidents and even loss of human lives (as demonstrated in Figure 1). Such critical faults can manifest in any part of a circuit. The structural design characteristics, along with a fault's location, play a crucial role in determining how critical these faults are, pertaining to the fact that the effectiveness of the fault propagation is closely tied to these structural features. Therefore, it is essential not only to assess the impact of faults, but also to consider their criticality in the context of their location within the design. This comprehensive approach is imperative for ensuring FuSa in mission-critical applications.

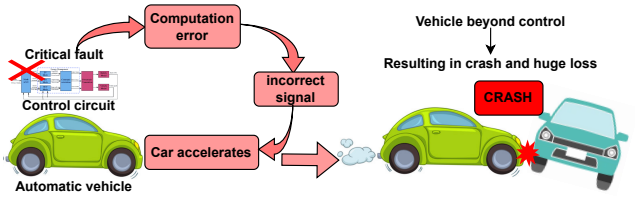


Figure 1. Detrimental effects of fault manifestations in E/E systems.

Traditionally, the evaluation of the reliability of E/E systems has relied on the methodologies of fault injection and simulation [7]. Nonetheless, these approaches face significant hurdles when confronted with the increasing complexity of designs, including substantial timing overheads. Moreover, these approaches heavily depend on resource-intensive processes, such as application specific workload generation, and fall short of identifying critical components in the design that could potentially result in functional failures. Considering the increasing complexity of modern designs, it becomes crucial not only to determine whether a circuit node is critical, but also to establish a systematic scoring system for quantifying its level of criticality. For example, while two nodes may both be considered critical, one may exhibit higher criticality, signifying a greater priority in fortification. Conventional fault analysis methodologies, unfortunately, fall short in this regard as well, necessitating the exploration of alternative approaches to fulfill this emerging need.

Towards this end, in this paper, we present a novel ML-based framework for identifying circuit nodes exhibiting a higher susceptibility to critical faults. Since graphs serve as a suitable data structure for encapsulating the relational aspects of an E/E circuit, our initial step involves the translation of circuit designs into graph structures. Following this, a Graph Convolutional Network (GCN) is utilized for the fault criticality analysis, since they are designed to operate on graph-structured data and consider both the node's intrinsic properties and those of its adjacent nodes during training and inference stages [12].

Our primary objective is to develop an ML-classifier that leverages traditional fault injection results obtained from a subset of the design and utilize the knowledge to categorize circuit nodes within the remaining part of the design, thereby mitigating the necessity for conventional fault injection procedures across the entire circuit.

*Corresponding author: Sanjay Das (Email: *sanjay.das@utdallas.edu*)
This work is supported by the Semiconductor Research Corporation (SRC GRC Task: 3192.001).

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3656498

This streamlined approach results in resource and time savings during fault criticality analysis. Following this, an explainability analysis is conducted with the objective of not only interpreting model predictions but also assisting in the identification of crucial design features (such as gate types and connections), elucidating their contribution to the criticality of circuit nodes during fault manifestations. Furthermore, to indicate the criticality levels of circuit nodes, we incorporate a regression analysis, which assigns criticality scores to nodes, post-classification. Thus, this approach provides a holistic view of a circuit's fault criticality landscape.

The key contributions of this work is as follows:

- We present an ML-based framework, namely a GCN, for critical node classification in a circuit. We generate a graph representation from the circuit and train a GCN model for identifying critical nodes in the design.
- We utilize a modified version of the GCN model to calculate node-specific criticality scores, revealing the range of criticality that circuit nodes pose in faulty scenarios.
- By incorporating explainability analysis, we interpret GCN model predictions and identify critical design features, contributing to the criticaility of circuit nodes during faults.
- When evaluated on state-of-the-art designs, our framework furnishes up to 93.7% classification accuracy in identifying critical nodes, thereby demonstrating the efficiency of the proposed technique.

The rest of the paper is organized as follows. Section 2 provides the background information. The proposed methodology is explained in Section 3. The efficiency of the proposed framework is demonstrated in Section 4. Finally, Section 5 concludes the paper.

2 Background

In this section, we shed light upon the essential background information for elucidating our ML-based approach to analyze fault criticality in E/E circuits.

2.1 Graph Convolutional Network

Graph Convolutional Networks (GCN) are a powerful ML-framework with applications ranging from social networking, traffic prediction to solving electronic design automation (EDA) problems necessitating processing of irregular data structures [5][10]. These networks assimilate information from the provided graph input, which is augmented by attributes associated with nodes and edges. These attributes are propagated across the nodes through message passing facilitated by edge connections during training. The primary objective is to acquire the capacity to learn a function reliant upon the features within a given graph denoted as $G = (V, E)$. A feature matrix X of dimensions $N \times F$, where N signifies the node counts within the graph and F represents the number of input features for each node, contains the structural information of the graph. An adjacency matrix A serve as a representation of the graph's structural characteristics by capturing the connection information between nodes. The model takes the feature matrix X and the adjacency matrix A as input and furnishes a matrix Y (=$N \times O$) as output, where O signifies node output features. Each layer within the GCN can be expressed as a non-linear function, as elaborated in Equation 1. Here, $H^{[i]}$, $H^{[i+1]}$ indicate the feature representations for layers i and $i+1$. W^i represent the weights in layer i and σ is the activation function. A^* is the normalized adjacency matrix of the input graph.

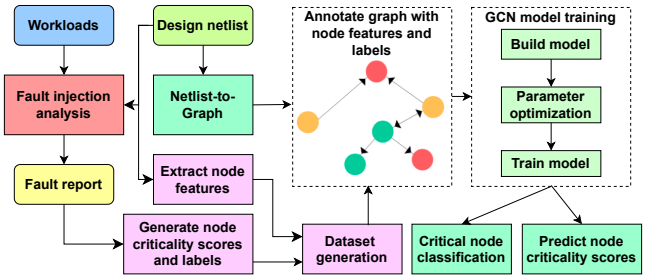


Figure 2. Overview of the proposed framework.

$$H^{[i+1]} = \sigma(W^i H^i A^*) \tag{1}$$

$$f(H^{(i)}, A) = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(i)}W^{(i)}) \tag{2}$$

To address constraints in adjacency matrix A, several steps are taken: adding the identity matrix (I) to A, introducing self-loops to the graph. Additionally, A is normalized to ensure that all row sums equal one, preserving feature vector scales. Typically, symmetric normalization is used for this purpose. Consequently, a GCN layer can be represented as Equation 2, which follows Equation 1 with A^* being replaced by the normalized adjacency matrix ($A^* = \hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}$).

2.2 Fault Injection (FI) Analysis

Fault injection (FI) is a widely employed technique for assessing circuit reliability and evaluating the potential for fault propagation, which can lead to errors and system failures. FI encompasses three primary categories: physical fault injection, simulation-based fault injection, and software fault injection [7]. To avoid the need for costly respins, it is crucial to apply fault injection-based reliability analysis as early as possible in the design process. Consequently, simulation-based fault injection emerges as the predominant method due to their cost-effective nature [1]. This approach, conducted at the netlist level, does not require any modifications to netlist source code. However, it is important to note that this approach can be exceptionally time intensive, as analyzing complex circuits exhaustively may necessitate a multitude of FI experiments and substantial simulation time. FI could potentially be expedited by harnessing ML algorithms to construct predictive models using empirical data. To this end, existing research propose an ML-based approach for accelerating simulation-based fault injection analysis [9]. However, these techniques focus solely on utilizing node attributes for predictions, disregarding essential structural information embedded in the design, including the connections between nodes. These structural elements are pivotal in understanding fault criticality and its effects on the overall design.

3 Proposed Methodology

In this section, we elaborate the proposed ML-based framework, for the classification of circuit nodes (as illustrated in Figure 2). First, the framework constructs a graph representation of the circuit. Following this, node features are extracted from the design. Subsequently, a dataset is built using fault injection experiments performed on the design. Next, a GCN model is trained by utilizing the graph information and the dataset. Finally, the trained model is employed for critical node classification and score prediction using regression. Moreover, an explainability analysis is performed to interpret the model predictions.

3.1 Graph Generation and Feature Extraction

In this procedure, a graph is constructed from the design netlist by systematically traversing the netlist and gathering pertinent information about each node (a gate in the netlist). The node features include structural and functional information about the node such as number of connections, intrinsic state probabilities, transition probabilities. These node features play a pivotal role in the training the GCN model. The node features are defined as follows:

3.1.1 Number of connections: The total count of *fan-ins* and *fan-outs* linked to a node or gate is designated as the number of connections. This attribute holds notable significance due to the fact that nodes exhibiting a high number of connections can be extremely critical in the event of faults, given the likelihood of fault effects spreading from said node to its connected components.

3.1.2 Intrinsic state probability. Intrinsic state probability denotes the likelihood of a gate's output being in a specific state (0 or 1). When a gate exhibits a strong tendency toward a particular logic state, faults causing state transitions have reduced impact. Conversely, gates with equal probability (~0.5) of being each state, render circuits more susceptible to faults, as they readily oscillate between states. Therefore, intrinsic state probability is a crucial attribute for assessing a circuit's vulnerability to faults.

3.1.3 Intrinsic transition probability: Intrinsic transition probability signifies the likelihood of the output of a node transitioning from one state to another, specifically, from 0 to 1 or vice versa in response to an input change. This attribute is crucial as it governs the likelihood of a gate altering its output state when subjected to faults at its inputs. A higher probability indicates an increased possibility of fault propagation, potentially causing functional errors.

3.1.4 Boolean tag, if gate negates logic: This tag indicates whether the gate logic performs a negation operation (for instance, a NAND gate would have a tag of 1, while an AND gate would have a tag of 0). This feature is crucial as negating gates are more robust to circuit faults due to having fewer transistors, and resiliency to noise owing to the inversion process.

3.2 Generating fault criticality dataset

In order to obtain the ground truth, for training the proposed GCN model, we establish a comprehensive dataset through FI simulations against diverse workloads.

3.2.1 Fault injection: In this step, faults are introduced to the design netlist, which are then subjected to workloads to assess their detection capability. At the onset, a faultless golden simulation of the design is generated for a workload, noting the functional outputs attained in the absence of any faults in the circuit. Next, faults, namely stuck-at-0 and stuck-at-1, are introduced into the design. The outcomes produced under these fault manifestations are then matched against the reference results, enabling the assessment of the severity of the fault.

3.2.2 Dataset generation: Following the fault injection analysis, a node-criticality dataset is formulated by utilizing fault injection reports developed in Section 3.2.1. These reports, originating from diverse workloads, are aggregated to derive a criticality score for each node. This score serves as an indicator of criticality level of a node under consideration. Subsequently, based on a pre-determined criticality score threshold, the nodes are categorized into two distinct groups, namely "Critical" and "Non-critical", as part of our node classification assessment. The determination of the fault criticality threshold is contingent upon the unique application context and the stipulated fault tolerance criteria, and therefore is upto the respective stakeholders. In the context of our experimental setup as described in section 4.1, we have selected a threshold value of 0.5 for criticality assessment. This score implies that a node is deemed critical if a fault in that node leads to functional errors for more than 50% of the time.

Algorithm 1 Dataset Generation

Require:
Gate-level design netlist $\mathcal{D}$;
Workloads;
Fault injection reports;
Number of workloads $\mathcal{N}$;
Criticality threshold th;

1: $NodeCritic \leftarrow emptyDict()$
2: $NodeLabel \leftarrow emptyDict()$
3: **for** $workload$ in $Workloads$ **do**
4: $Report \leftarrow FaultInjection(D, workload)$
5: **for** $node, label$ in $Report$ **do**
6: **if** $label$ == "$Dangerous$" **then**
7: $NodeCritic[node] \leftarrow NodeCritic[node] + 1$
8: **end if**
9: **end for**
10: **end for**
11: **for** key in $NodeCritic.keys()$ **do**
12: $NodeCritic[key] \leftarrow NodeCritic[key]/N$
13: **if** $NodeCritic[key] \geq th$ **then**
14: $NodeLabel[key] \leftarrow 1$
15: **else**
16: $NodeLabel[key] \leftarrow 0$
17: **end if**
18: **end for**
19: **return** : $NodeCritic, NodeLabel$

Algorithm 1 illustrates the process of generating our experiment dataset. First, two empty dictionaries (NodeCritic, NodeLabel) are instantiated for capturing the node criticality scores and labels, respectively (*lines 1-2*). Next, the fault simulation experiments are performed and the reports are generated (*lines 3-4*). Subsequently, the criticality categorization of each node is aggregated in a dictionary (*lines 5-10*). Following this, the aggregated criticality score of each node is compared to a pre-defined threshold value (th) and the nodes with the same or higher criticality are labeled 1 or 0 otherwise (*lines 11-17*). Finally, the criticality scores and labelled nodes are provided as the output (*line 18*).

3.3 GCN Model Training

This section describes how we build and optimize the GCN model for critical node classification and criticality score prediction.

3.3.1 Model construction: The graph built in Section 3.1, accompanied by the node features, criticality scores and labels, serves as the basis for training a GCN model, consisting G_l graph convolution layers. The input layer has F neurons, where F is the feature dimentionality of the graph nodes. We define the model as a function $GCN(x)$ producing output $y \in R^2$ for input $x \in R^n$. The output of $GCN(x)$ is a 2-dimensional vector containing the probabilities P_c of classes ($C \in \{0, 1\}$). The final prediction class y_c is obtained by applying the $argmax$ function: $argmax(GCN(x)) = y_c$.

Algorithm 2 Graph Convolutional Network Development

Require:
Gate-level design netlist $\mathcal{D}$;
1: $G(V, E) \leftarrow GetGraph(D)$
2: $A_{N\times N} \leftarrow AdjacencyMatrix(E)$
3: $Features \leftarrow ExtractFeatures(G, D)$
4: $X_{N\times F} \leftarrow FeatureMatrix(Features)$
5: **for** v in V **do**
6: $e_v \leftarrow E_f(v)$
7: **end for**
8: **for** l in layers L **do**
9: $e_v \leftarrow AggregateNeighbours(e_v, A_{N\times N}, W_i)$
10: $e_v \leftarrow ActivationFunction(e_v)$
11: **end for**
12: $e_v \leftarrow AggregateNodeFeatures(e_v, X_{N\times F}, W_f)$
13: $e_v \leftarrow ActivationFunction(e_v)$
14: $Y_c \leftarrow logSoftmax(W_y * e_v)$

3.3.2 Hyperparameter Optimization: To determine the optimal architecture for our GCN model, we employ a grid search algorithm to find the best hyperparameter set. This involves sweeping through parameters like the number of layers, layer types, and input-output feature dimensions within the GCN.

3.3.3 Model training: We train the GCN model using the circuit's graph representation, node features, edge connections, and the labeled dataset created in Section 3.2. Training involves using a subset of design nodes, while the remaining nodes are reserved for validation post-training. The GCN training process is detailed in Algorithm 2, which utilizes the graph (G) generated from the design netlist (D) as the input argument. First, the edges and node features are extracted from the design (*lines 2-3*). The extracted features are represented as a matrix of size $N \times F$, where N is the number of nodes and F is the number of features (*line 4*). Subsequently, the node embeddings for each node (v) is initialized using an embedding function E_f, which results in embeddings e_v (*lines 5-7*). Next, in each iteration of message passing, the embeddings of each node v and that of its neighbours are aggregated using the matrix A and a weight matrix W_l. The resulting vector is passed through an activation function, and the updated embedding e_v is stored for the next iteration (*lines 8-11*). Following the last iteration, the final node embeddings are computed by aggregating the original node features X, and a weight matrix W_f. These final embeddings can be subsequently utilized for node classification (*lines 12-14*), where the class probabilities Y_c are calculated by passing the final embeding e_v through an activation function *log_softmax*. The class label can be obtained by applying *argmax* function on the class probabilities.

3.4 Node Criticality Estimation

To evaluate the node classification performance by the trained GCN model, a set of unseen nodes is applied to the model along with their respective features for generating class predictions. Following the categorization of circuit nodes into critical and non-critical groups, a regression analysis is conducted to estimate their respective levels of criticality. This process produces continuous scores ranging from 0 to 1, where 0 signifies the lowest and 1 the highest criticality. For example, when two nodes are labeled as "critical", the classifier does not differentiate between them. Conversely, in regression analysis, it is possible for two such nodes to have unique criticality scores, such as 0.55 and 0.75, which are useful in distinguishing the levels of criticality they pose during faults. To predict these criticality scores, we slightly modify to the GCN classification model. We remove the *logSoftmax* activation function from the output layer and change the output dimensionality from 2 to 1, enabling the generation of continuous regression scores for criticality prediction. This modified model is trained over multiple epochs and then deployed for predicting node criticality scores.

3.5 Interpreting predictions through GCN Explainability

In this step, our principal objective is to furnish meaningful interpretations for the predictions generated by our GCN model concerning critical node classification. Towards effectively achieving this objective, we have adopted a model-agnostic post-hoc approach, GNN-Explainer, which have demonstrated its efficacy in literature [13]. This approach aims to construct a subgraph $G_s \subseteq G$ (the Control Data Flow Graph) and the associated node features $X_s = x_i | v_i \subseteq G_s$, emphasizing their critical contribution to the node's classification by the model. Once the GCN model is trained, it is employed to generate predictions for sets of nodes, not encountered during the training process. To interpret these individual predictions, the GCN model, the feature matrix, the respective target node, and the number of iterations, are used to build an explainer object. Subsequently, the explainer generates importance scores for each feature and the corresponding subgraph, offering insights into the classification of a node as either "Critical" or "Non-critical". The individual node explanations of all the circuit nodes are analyzed to the develop a global feature importance map for the entire model. Towards this end, we aggregate the node feature scores for all individual predictions. Similarly, we rank the features based on the feature scores and aggregate the rankings for all the nodes (refer to Equation 3, where n is the number of nodes).

$$Avg_{FeatureRank} = \frac{1}{n}(\sum_{i=1}^{n} FeatureRank(node_n)) \tag{3}$$

Through the evaluation and ranking of these feature scores, we discern the significance of the top-ranked features and their influence on the overall performance of the model.

4 Experimental Evaluation

4.1 Experimental setup

The proposed GCN framework is evaluated by utilizing three designs: an SDRAM controller and two modules (Instruction Fetch (IF) and Instruction Cache Finite State Machine (ICFSM)) from the OR1200 architecture. Considering the substantial variations of these designs, they present a robust basis for evaluating our proposed framework. To create our dataset, we first map an register-transfer level (RTL) design to a gate-level synthesized netlist using Synopsys design vision. Next, we utilize the Cadence Xcelium tool

Table 1. GCN Network configuration.

Layer	Type	Parameters		Values
		In	***Out***	
1	Graph convolutional layer	Input	16	-
2	Rectified Linear Unit	-	-	-
3	Graph convolutional layer	16	32	-
4	Rectified Linear Unit	-	-	-
5	Dropout Layer	-	-	0.3
6	Graph convolutional layer	32	64	-
7	Rectified Linear Unit	-	-	-
8	Graph convolutional layer	64	2	-
9	Log Softmax	2	2	-

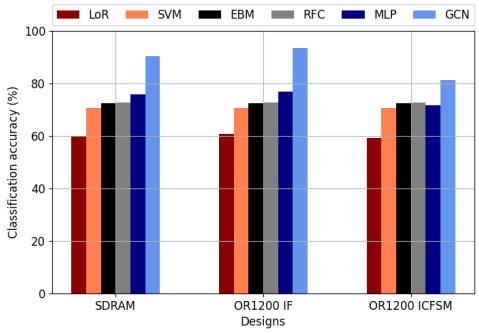


Figure 3. Critical node classification accuracy for all three designs using several ML-techniques.

for fault injection experiments [3]. This tool injects faults into the design netlist and conducts circuit simulations using various input vectors (workloads). Subsequently, detailed fault detection reports are generated, capturing fault criticalities and detection coverage under different workloads. Using these fault reports, we compile a comprehensive dataset, containing information on node criticality during faults. This data is then used to compute node criticality scores, which are crucial for labeling nodes as either "Critical" or "Non-Critical." This dataset forms the basis for training and evaluating the GCN framework. For each design, we partition the dataset into an 80-20 split for training and validation, respectively.

Performance Metrics: Model performance is assessed using various metrics: accuracy, the receiver operating characteristic (ROC) curve, and the area under the ROC curve (AUC). Accuracy calculates the percentage of correct predictions among all predictions. The ROC curve plots true positive rate (TPR) against false positive rate (FPR). TPR represents the accuracy of correctly identifying class 1 (critical) predictions, calculated by dividing the number of correct class 1 predictions by the total of correct class 1 predictions and incorrect class 0 predictions. Conversely, FPR indicates the rate of incorrectly identifying class 1 (critical) predictions, determined by dividing the number of incorrect class 1 predictions by the total of incorrect class 1 predictions and correct class 0 predictions. AUC, or Area under the ROC Curve, quantifies the total area beneath the ROC curve. Higher AUC values signify better model performance and the value ranges between 0 and 1.

The GCN model is implemented using PyTorch with the torch geometric package and constructed according to the specifications detailed in Table 1.

4.2 Experimental Results

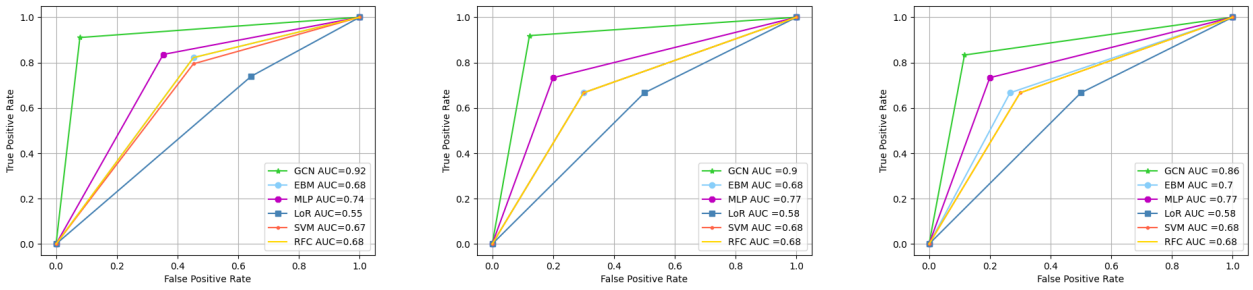
4.2.1 Node classification: In this experiment, we employ the trained GCN model to classify circuit nodes in the aforementioned designs. First, we elaborate on the design complexities of the circuits, following which we validate the trained model and compare its performance with existing ML-techniques, such as MLP, LoR, Random Forest Classifier (RFC), Support Vector Machines (SVM), and Explainable Boosting Machine (EBM) [5]. Figure 3 presents the classification performance of all the ML-models along with our proposed approach when evaluated on the same workloads. The corresponding ROC curves are plotted in Figure 4.

Case 1: SDRAM controller: A SDRAM controller serves as an interface that manages the communication between the computer's central processing unit (CPU) and the SDRAM memory modules. The trained GCN achieves an accuracy of 90.34% in critical node classification for the design. Existing classifiers furnish up to 77% classification accuracy as demonstrated in Figure 3, thus, providing, significantly degraded results compared to our proposed GCN model. Moreover, Figure 4a illustrates that the GCN model provides the best ROC curve with the highest AUC score of 0.92, demonstrating the efficacy of the proposed approach.

Case 2: OR1200 IF module: The OR1200 IF module is accountable for providing the processor with instructions to be executed. It consists of an instruction cache and the control logic to calculate the address of the instruction to be fetched. The trained GCN furnishes 93.7% accuracy for node classification. In comparison, the existing classifiers furnish upto 78% accuracy (Figure 3), thus, providing, significantly worse results compared to the GCN model. Moreover, Figure 4b illustrates that the GCN model provides the best ROC curve with the highest AUC score of 0.9.

Case 3: OR1200 ICFSM module: The OR1200 ICFSM module constitutes an essential component of a cache controller that furnishes all the signals to a processor, data array, and the primary memory. The state machine guarantees that the signals are established at the appropriate timings. The trained gcn classifier achieves an accuracy of 81.03% in identifying critical nodes. The existing classifiers furnish upto 72% accuracy (Figure 3), which is significantly lower compared to our proposed GCN-based approach. The GCN model also provides the highest AUC score of 0.86 (Figure 4c).

Interpreting GCN predictions: The GCN model predictions are interpreted in terms of feature importance scores as illustrated in Figure 5(a), where the feature scores for a randomly selected

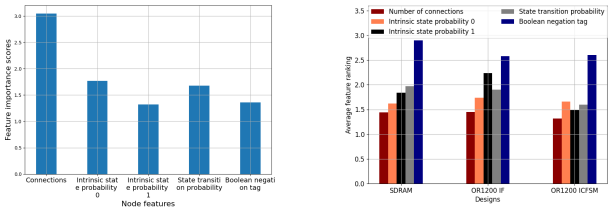


(a) SDRAM Controller. **(b)** OpenRISC 1200 IF module. **(c)** OpenRISC 1200 ICFSM module.

Figure 4. ROC curve to visualize the performance of various classifiers.

Table 2. Critical node classification along with corresponding feature importance scores and criticality score predictions.

Design	Nodes	Criticality classification	Feature importance scores					Criticality score
			Number of connections	Intrinsic state probability of 0	Intrinsic state probability of 1	State transition probability	Boolean inverting tag	
SDRAM Controller	ND2_U393	Non-critical	2.24	2.11	0.83	1.12	1.05	0.41
	AO3_U373	Critical	1.91	2.05	1.49	1.93	1.30	0.56
	AO2_U419	Non-critical	5.96	3.61	1.74	2.34	2.47	0.29
	ND4_U233	Critical	2.25	2.57	1.14	1.40	1.09	0.7
OR1200 IF module	IV_U112	Non-critical	2.12	1.92	0.94	1.24	0.88	0.44
	AO3_U143	Critical	1.97	2.05	1.14	1.34	1.05	0.61
	AO2_U194	Non-critical	2.56	2.25	1.06	1.16	0.98	0.41
	NR4_U129	Critical	2.26	2.65	1.27	1.32	1.08	0.56
OR1200 ICFSM module	ND2_U121	Critical	1.85	2.14	0.87	1.06	1.28	0.51
	AO3_U156	Critical	2.54	2.46	1.34	1.29	0.88	0.7
	NR2_U211	Non-critical	4.16	3.04	1.12	2.08	2.02	0.34
	NR4_U165	Critical	2.24	2.46	1.04	1.26	0.89	0.55



(a) (b)

Figure 5. (a) Feature importance scores for a graph node predicted by the GCN model and (b) the graph depicting the aggregated feature rankings for all three designs.

node (from the SDRAM controller design) is captured. Notably, the features "Number of Connections" (with score of 3.06) and "Intrinsic State Probability of 0" (with score of 1.75) emerge with the higher scores, indicating their pronounced significance in the prediction process. The feature importance scores for several nodes from all three designs are captured in Table 2. To comprehensively understand the relevance of various node features in shaping the overall behavior of the model, we rank the feature importance scores and aggregate their respective rankings for all node explanations ass depicted in Figure 5(b). The figure, consolidating results for all three designs highlight that "Number of Connections" and "Intrinsic State Probability of 0 and 1" are consistently ranked as crucial features contributing to the model's decision in node classification.

4.2.2 Node Criticality Score Prediction: Following the classification of nodes, we employ the GCN regressor to predict their respective criticality scores within each design. To maintain brevity, we have randomly sampled a limited number of nodes from each design and documented their class assignments, feature importance scores and the predicted criticality scores in Table 2. Notably, the predicted scores align with the classification outcomes, as demonstrated in the table. For example, within the SDRAM controller design, node ND4_U233 has been classified as "critical", and the corresponding criticality score predicted is 0.7, surpassing the defined threshold of 0.5. It is to be noted that these score predictions are not confined solely to the presented instances but extend uniformly across all nodes within all three designs with highconformity with the classification model. Therefore, our framework not only excels in the classification of circuit nodes, but also offers a mechanism for quantifying criticality scores.

5 Conclusion

This paper introduces an ML-based framework aimed at expediting fault injection analysis in E/E systems. The framework constructs a graph, representing the gate-level netlist of a design. The graph nodes are assigned criticality scores and labels based on fault simulation results. Subsequently, a GCN model is trained on a portion of the circuit and tested on the remaining nodes in the design. Upon extensive evaluation using large commercial circuits, the proposed framework furnishes up to 93.7% accuracy in classifying critical nodes. Following this, node criticality scores are generated using a regression analysis, post-classification and the predictions exhibit significant (over 85%) correlation with the predicted class. Furthermore, by incorporating an explainability analysis, the framework successfully identifies crucial design features, that influences fault criticality in the design. Hence, this study provides a thorough comprehension of the circuit's criticality landscape, offering valuable insights for improving robustness of hardware designs.

References

[1] Juan Carlos Baraza et al. 2005. Improvement of fault injection techniques based on VHDL code modification. In *Tenth IEEE International High-Level Design Validation and Test Workshop, 2005.* IEEE, 19–26.

[2] John Birch et al. 2013. Safety cases and their role in ISO 26262 functional safety assessment. Springer, 154–165.

[3] Cadence. 2023. Xcelium Fault Simulator. https://www.cadence.com/en_US/home/training/all-courses/86246.html, Last accessed on 2023-09-10.

[4] Arjun Chaudhuri et al. 2020. Functional criticality classification of structural faults in AI accelerators. In *2020 IEEE International Test Conference (ITC).* IEEE, 1–5.

[5] Arjun Chaudhuri et al. 2021. Fault-criticality assessment for AI accelerators using graph convolutional networks. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE, 1596–1599.

[6] ISO. 2011. 26262: Road vehicles-Functional safety. *ISO/FDIS* 26262 (2011).

[7] David Kammler et al. 2009. A fast and flexible platform for fault injection and evaluation in verilog-based simulations. In *2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement.* IEEE, 309–314.

[8] T Kogan et al. 2018. Advanced Uniformed Test Approach For Automotive SoCs. In *2018 IEEE International Test Conference (ITC).* IEEE, 1–10.

[9] Li Lu et al. 2021. Machine Learning Approach for Accelerating Simulation-based Fault Injection. In *2021 IEEE Nordic Circuits and Systems Conference (NorCAS).* 1–6. https://doi.org/10.1109/NorCAS53631.2021.9599646

[10] Yuzhe Ma et al. 2019. High performance graph convolutional networks with applications in testability analysis. In *Proceedings of the 56th Annual Design Automation Conference 2019.* 1–6.

[11] V Prasanth et al. 2021. Exploiting Application Tolerance for Functional Safety. In *IEEE International Test Conference (ITC).* IEEE, 399–408.

[12] Felix Wu et al. 2019. Simplifying graph convolutional networks. In *International conference on machine learning.* PMLR, 6861–6871.

[13] Zhitao Ying et al. 2019. Gnnexplainer: Generating explanations for graph neural networks. *Advances in neural information processing systems* 32 (2019).