**QUEEN'S
UNIVERSITY
BELFAST**

Estimating Code Vulnerability to Timing Errors via Microarchitecture-Aware Machine Learning

Tompazi, S., Tsiokanos, I., Martinez-del-Rincon, J., & Karakonstantis, G. (2021). Estimating Code Vulnerability to Timing Errors via Microarchitecture-Aware Machine Learning. *IEEE DESIGN & TEST OF COMPUTERS*.
<https://doi.org/10.1109/MDAT.2021.3135318>

Published in:
IEEE DESIGN & TEST OF COMPUTERS

Document Version:
Peer reviewed version

Queen's University Belfast - Research Portal:
[Link to publication record in Queen's University Belfast Research Portal](#)

Publisher rights
Copyright 2021, IEEE.
This work is made available online in accordance with the publisher's policies. Please refer to any applicable terms of use of the publisher.

General rights
Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The Research Portal is Queen's institutional repository that provides access to Queen's research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person's rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact openaccess@qub.ac.uk.

Estimating Code Vulnerability to Timing Errors via Microarchitecture-Aware Machine Learning

Styliani Tompazi, Ioannis Tsiokanos, Jesus Martinez del Rincon and Georgios Karakonstantis

School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, UK

Email: {stompazi01, i.tsiokanos, j.martinez-del-rincon, g.karakonstantis}@qub.ac.uk

Abstract—The adoption of aggressively down-scaled voltages along with worsening process variations render nanometer devices prone to timing errors that threaten system functionality and output quality. In this paper, we introduce a significance-aware code vulnerability factor (SCVF) for early evaluation of the impact of such errors on applications. To estimate this metric, we propose the utilization of a microarchitecture-aware ML model for timing error prediction that jointly considers many instruction types, as well as all in-flight instructions in a pipeline. Finally, we validate the efficacy of SCVF by analyzing the output quality loss of various applications.

Index Terms—Timing error modelling, machine-learning, microarchitecture, dynamic timing analysis, application error vulnerability

1 INTRODUCTION

The continuous pursuit for miniature, energy-efficient devices through the aggressive scaling of transistor sizes, and supply-voltages has made circuits very sensitive to timing errors [1]. Modelling such errors and estimating the vulnerability of programs to them is extremely challenging since their manifestation depends on various static (i.e., microarchitecture-aware), as well as dynamic (i.e., data-dependent) parameters [2].

The most accurate way to capture timing errors in design cycle is through gate-level simulation (GLS), however, it is extremely slow. Recent approaches have tried to capture most of such parameters and develop timing error models using complex statistical and machine learning (ML) methods [3]–[6]. Nevertheless, these models have neglected some important microarchitecture features and were not used to estimate the susceptibility of any code to timing errors early at the design cycle to help designers choose appropriate design options. Only few existing studies tried to estimate the vulnerability of applications to errors by evaluating different metrics [7], [8]. Typically, such metrics are leveraged to evaluate either the impact of errors on applications or their reliability. Although very interesting, these studies require extensive time-consuming error-injection campaigns and quantify the vulnerability only according to the total number of errors. Such methods overlook the fact that timing errors, depending on their location, may have different effects on the output quality, thus hindering the evaluation of application error-resilience [7]. Failing to understand the timing error effects on applications may guide wrong design decisions and nullify the efficacy of existing error mitigation techniques.

Contributions. This paper, presents a novel application of ML on the microarchitecture-aware modelling of timing errors and the estimation of code vulnerability to errors through a new metric that considers the error location and

significance. Our contributions are summarized as follows:

- We quantify the degree to which various applications are prone to timing errors by introducing a new significance-aware code vulnerability factor (SCVF). Unlike existing vulnerability metrics that rely on exhaustive error-injection campaigns and consider only the total number of manifested errors [7], [8], SCVF utilizes a ML-based error prediction model which is able to consider the exact time (i.e., instruction) and location (i.e., bit) of a timing error. This allows us to capture the significance of those errors to the output quality, without relying on long simulations.
- We develop a workload-aware error prediction model based on supervised ML methods. We incorporate, for the first time, more complex microarchitecture features during the execution of various instructions in a pipelined core, such as the instruction execution history (i.e., order and type of instructions within a pipeline at any instant) into timing error modelling. Our model jointly considers several instruction types, rather than separate models for each individual instruction [5], [9], to predict the exact error location accurately, allowing the interpretation of error significance. Our results verify that deeper execution history leads to up to 11% accuracy improvement when compared to existing works that considered only one previous instruction [5], [9].
- We validate the efficacy of SCVF by comparing the error-induced quality loss obtained by the ML prediction model and the quality loss observed for the instruction sequences (ISQs) that we extract from a representative set of benchmarks. The ML-based metric allows us to perform a quick and accurate evaluation of error vulnerability by eliminating the need for time-consuming error-injection and GLS campaigns.

2 BACKGROUND & RELATED WORK

Typically, a pipelined, computational core consists of a set of N unique timing paths $P = \{P_1, P_2, \dots, P_i, \dots, P_N\}$, which are characterized by their delays $D(p_i)$. Each of these paths can be found within exactly one pipeline stage $s = 1, 2, \dots, S$ and only a few of them will be excited at every instance depending on the executed instruction. The main fields of a data-processing instruction are an operation code (OP) and two input operands (ORa and ORb) represented as a binary vector with k bits. Figure 1 provides an example of a pipelined design, where each stage processes a specific part of one instruction at a time, allowing the parallel execution of multiple instructions. By the terms parallel or concurrent execution, we mean that up to S instructions share the same hardware circuitry (i.e., pipeline) in a time-sharing fashion. Program flow of Figure 1 includes a set of N instructions $I = \{I_1, \dots, I_N\}$ which are executed in sequences ISQ_n , each consisting of d instructions such that $\bigcup_{n=1}^{N/d} ISQ_n = I$. The number of instructions within a sequence depend on the pipeline depth $d = 1, 2, \dots, N$ where N denotes the maximum pipeline depth. Each pipeline depth ties the circuit's behavior on the current instruction i.e., the latest instruction entered the pipeline to the history of all the preceding instructions in the pipeline. In this example, I_3 as the currently executed instruction that is in-flight with I_1 and I_2 .

As in any synchronous design, the conventional static timing analysis (STA) determines the clock period (CP_{STA}) at design time, such as:

$$CP_{STA} = \max_{p \in P} \{D(p)\} \quad (1)$$

During circuit operation, only a few paths of the overall path set will be excited depending on the processed instruction and its operands. Any excited path P_i has a positive timing slack until the so-called point of failure (PoF). In case of any delay increase or clock reduction, which exceeds the available slack the activated path P_i will fail since $D(P_i) > CP_{STA}$, leading to a setup timing error.

2.1 Dynamic Parameters Affecting Timing Errors

The possibility of a path to fail under any delay increase depends on many dynamic parameters, such as the type of the in-flight instruction and its operands, as well as the instruction execution history within the pipeline.

2.1.1 Instruction Type & Input Operands

In each clock cycle, both operands and especially the type of instruction that is currently in-flight in the pipeline stage s determines the delay of the activated path and thus the number and the distribution of timing errors. Instructions, which activate critical long paths tend to fail more frequently. Moreover, depending on input operands, the same type of instructions may activate different paths with various latency requirements, leading to different error rates [5].

2.1.2 Instruction Execution History

Pipelined execution of instructions also influences the occurrence of timing errors. This is because concurrently executing instructions share control signals and execution stages,

Fig. 1: Instruction sequences (ISQs) and delay requirements across different paths and stages in a 3-stage ($d=3$) pipelined core.

affecting the state of the forwarding logic, and thereby place great demand on circuit timing deadlines. That is, the joint effects of the in-flight instruction and a number of previously executed instructions determine the delay of the excited path in every pipeline stage and thus the error rate.

2.2 Error Modelling & Error Vulnerability Estimation

To model the dynamic timing behaviour, instruction-aware error models have been proposed [3]. They statistically determine the output bit failure probability by performing dynamic timing analysis (DTA) based on several instructions with randomized operands. Despite their statistical nature, such frameworks suffer from inaccuracies because they neglect the influence of previously executed instructions on error manifestation.

Subsequent ML-based models [5], which studied the correlation between instruction history and errors, indicated that besides the currently executed instruction, only the instruction in the previous cycle affects timing errors. However, such an observation holds only in the case of simple, non-pipelined, functional units. A recent study [10] indicated that the order and type of instructions within sequences that have depth equal to the pipeline depth significantly affect the error rate. Such a work paved the way to incorporate instruction execution history into the timing error modelling, but it did not use instruction execution history to train a ML model for predicting bit-level timing errors, as we do in this paper. Although [10] presents a very interesting approach, this framework is operand-agnostic and thus, overlooks the impact of input operands on timing error manifestation. As a result, [10] is unable to predict the exact bit location of timing errors.

Existing studies [7], [8] have introduced metrics to help evaluate the impact of errors by formulating vulnerability metrics, such as the application vulnerability factor (AVF). However, such metrics rely on exhaustive error-injection campaigns that use single bit-flip models. Those models might be accurate for evaluating soft errors, but are inadequate in timing error modelling, where many bits can fail depending on the input data and circuit state. These metrics focus on the impact of errors at the architecture level rather than the application level. In fact, the proposed metrics are associated with the total number of bits that cause incorrect

output or irregular behaviour, and thereby, overlook the significance of the error and its impact on the output quality.

2.3 Motivation

Undeniably, modelling the inherent complex dynamic timing behaviour of pipelined units considering all possible parameters that affect dynamic timing errors (clock reduction, type of instructions, input operands, instruction execution history) is very challenging. Particularly, a pipelined design, which executes several types of instructions, each of them consisting of two input operands (ORa and ORb), has the following number of possible combinations of execution parameters (Exp):

$$Exp = (\#OPs \times 2^{ORa_bits} \times 2^{ORb_bits})^{pipeline_depth}$$

For instance, in the floating-point unit under test (see Section 4), there are 6 instruction types and thus 6 different OPs, with two double precision 64-bit instruction operands and a pipeline depth of 6 stages. Thus, the number of Exp is:

$$Exp = (6 \times 2^{64} \times 2^{64})^6 \approx 7.24 \times 10^{235}$$

Exhaustively evaluating all these parameters is extremely time-consuming, if not practically infeasible. This motivates our work on properly formulated ML algorithms for considering relevant parameters for accurate timing error modelling.

Furthermore, estimating the resilience of a program to errors is a challenging task. An applicable metric should not only measure the probability of the error occurrence, but also acknowledge the fact that some errors have greater impact on the performance of an application than others. However, the utilization of typical output quality loss metrics (e.g., absolute error, mean squared error) require complex simulations on slow tools.

3 PROPOSED APPROACH

To address the above challenges, we defined a metric that estimates the error induced quality loss of an application, while acknowledging the error significance. To calculate this metric, we developed a framework that properly formulates and evaluates several ML algorithms, which jointly consider the data-dependent path activation and execution history of various instructions. The considered execution history depends on the target design. In contrast to existing error prediction models, the developed microarchitecture-aware model is able to predict timing errors at bit-level across different types of instructions, removing the need for different models for each individual operation as needed by existing frameworks [5], [9].

The overall workflow of the proposed approach is depicted in Figure 2. Note that the Model Training phase is being executed only once for developing the ML model, using each of the considered ML algorithms; while Model Evaluation phase runs each time we want to evaluate our model. Profiling & Timing Error Extraction phase examines the manifestation of timing errors and provides inputs to the Model Training and Model Evaluation phases.

In this study, we focus on floating point operations, as they have been proven to be amongst the most prone

Fig. 2: Workflow of the proposed approach.

to timing errors [4], [10], [11]. Nonetheless, our work can be applied to any other design and operation since the proposed workflow is implemented in a fully automated way (see Figure 2).

3.1 Application Error Vulnerability

Our tool characterizes the application vulnerability to errors by inspecting their binary code. To achieve this, we utilize the bit-level predictions of our ML-based error model, which classifies each bit as erroneous ($C_i = 1$) or error-free ($C_i = 0$).

To quantify the error vulnerability of each application based on error significance, we introduced a new metric, the SCVF, defined as:

$$SCVF = \frac{1}{\#ISQs} \cdot \sum_{n=0}^{\#ISQs} \sum_{i=0}^K \frac{C_i \cdot 2^i}{2^K - 1} \quad (2)$$

where $\#ISQs$ indicates the total number of ISQs evaluated for an application, C_i is the predicted output class for the bit in position i and K is the total number of predicted bits per ISQ. Essentially, the SCVF estimates the average value of the ratio of error significance over the maximum possible error for a given ISQ set (which represents a program).

3.2 Model Formulation and Development

We develop a model that classifies each bit of the executed instruction as erroneous ($C1$ class) or error-free ($C0$ class), thus implementing a multi-output binary classification model.

Training data. We gather training data by profiling real-world applications (by simulating the design under 25% of delay increase) mentioned in Section 4. Then, we randomly choose 1M ISQs from those applications. Additionally, we generate 500K random ISQs, which are combined with the ISQs extracted from the applications. To synthesize random ISQs, we design a python-written data generator, while for collecting real ISQs, we use an open-source profiling tool. We apply this tool to instrument program binaries that we run on real hardware and extract ISQs for each benchmark. Each ISQ consists of d instructions.

Input Features. We present ISQs in a binary format as follows: $\{OP(t-d+1), \dots, OP(t), OR_a(t-d+1), OR_b(t-d+1), \dots, OR_a(t), OR_b(t)\}$, where t denotes the current clock cycle; $OP(t)$ is the opcode of an instruction executed at the clock cycle t , consisting of 6 bits; $OR_a(t)$ and $OR_b(t)$ are the 64-bit operands of the instruction executed at the clock cycle t . As each bit value can affect the dynamic path sensitization and, therefore the timing class, we use each independent bit as an input feature. Thus, an input sample to the ML algorithm consists of the combination of d pipelined instructions, forming an array of binary features, with a size of:

$$\#features = d \times (6 + 2 \cdot 64) = d \times 134 \quad (3)$$

(bits) per input sample in total.

Labels. To assign labels to the training data, we use the Profiling & Timing Error Extraction phase depicted in Figure 2. This phase implements the typical ASIC flow, including Synthesis and Place and Route steps, and performs dynamic timing analysis (DTA) on a targeted core to identify the actual dynamic timing errors. Provided that every input data set under nominal conditions produce an error-free output, we estimate timing errors under any delay increase by comparing the simulation output with the error-free output. We represent potential degrees of delay increase by appropriately reducing the clock period [5], [10]. Note that we define the output of the applied DTA as golden output.

We train a single model that simultaneously predicts all bits. For a k -bit output, we have $C_i[t]$ output class labels, where $i \in [0, k-1]$ indicates the bit position. $C_i[t] = 0$ implies an error-free bit, while $C_i[t] = 1$ suggests an erroneous bit.

Training Process. In order to choose an appropriate ML method, we evaluate multiple supervised algorithms, such as Logistic Regression (LR), Support Vector Machine (SVM), Random Forest (RF) and Multi-Layer Perceptron (MLP), using the scikit-learn framework of python, with default configurations. MLP is a shallow neural network, consisting of a single hidden layer of 100 neurons, followed by a fully connected layer. The labels of the training and testing set for each predicted bit is not balanced, as some bits tend to fail less often.

3.3 Evaluation

To calculate SCVF we use the prediction output of the proposed ML model. We complete the evaluation of SCVF by comparing its efficiency with the actual output quality loss, which is measured using DTA.

To evaluate the proposed error model we use the following metrics that are commonly used in ML:

- $Accuracy = \frac{\#correct\ predictions}{\#total\ predictions}$
- $TPR = \frac{\#correct\ error\ predictions}{\#total\ error\ predictions}$
- $TNR = \frac{\#correct\ error\ free\ predictions}{\#total\ error\ free\ predictions}$

Although *Accuracy* (*Acc*) is a means of explaining how well a method predicts the correct class of a test set, it can usually be misleading, as the test set can be highly imbalanced. Thus, the evaluation of other ML metrics is

essential. *TPR* and *TNR*, provide a complementary view of the performance of our approach even under imbalance data as opposed to *Acc* [5]. Note that existing relevant studies have only considered *Acc* which may be ambiguous [5], [9].

We compare the output of our model to the golden output provided from DTA and state-of-the-art approach where $d = 2$ [5], [9]. Note that we compare our final model with our 64-bit model where $d = 2$, as state-of-the-art approaches [5], [9] have multiple models, targeting a single instruction type each time, while we combine multiple instruction types in a single model.

In addition, we are evaluating the inference time by measuring the prediction time required per sample.

Testing data. The testing dataset is acquired similarly to training data, under an assumed 25% delay increase. It consists of 340K randomly selected ISQ from the same set of applications, which have not been revealed to the classifier during the training process. We complete the evaluation of our model by performing cross-application testing on four real-world applications.

4 EXPERIMENTAL RESULTS

4.1 Experimental Setup

To showcase our approach, we consider an IEEE compatible FPU with a pipeline depth of 6 stages (i.e., $d = 6$). This unit is part of the mor1kx MAROCCHINO pipeline, which is an out-of-order processor implementing the OpenRISC 1000 instruction set architecture. The following double-precision operations are supported: addition, subtraction, multiplication, division, integer to floating-point and floating-point to integer conversion, which are among the most sensitive instructions to timing errors [5], [10].

In this work, we focus on timing errors resulting from clock reduction [5], [10]. This technique emulates worst-path delay increase from different possible sources of delays (e.g., process variations, transistor aging). For our experiments, we reduce the clock period by 25% which is consistent with the levels of variation-induces delay increase that have been reported in literature [2], [9], [10].

4.1.1 Circuit-level Conditions

The target FPU is implemented using the CCS NanGate 45 nm cell library. Synthesis and Place and Route steps are conducted using Synopsys Design Compiler and Cadence Innovus, respectively. The maximum clock frequency achieved is 400 MHz (i.e., clock period = 2.5 ns). As for the post-layout gate-level simulation, we use ModelSim from Mentor Graphics.

4.1.2 Application Profiling & Model Evaluation

To extract floating-point ISQs, and collect training and testing samples for the ML models from real-world applications we use DynamoRIO, an open-source profiling tool.

Specifically, we extract training and testing data, as described in section 3, from the following programs of the NAS benchmark suite: the BT, CG, IS and LU programs. Additionally, we extract ISQs from the Hotspot, Kmeans, Sobel and Srad programs to perform a cross-application evaluation of our model. This set of benchmarks represents a variety of algorithms that have many floating-point

Classifier	avg Acc	avg TPR	avg TNR
LR	73.2%	82.2%	71%
SVM	57.8%	3.3%	96.5%
RF	77.6%	90.3%	58.8%
MLP	77.5%	79.8%	63.2%

(a) ML algorithms comparison (d=2)

(b) RF performance evaluation vs instruction execution history depth (d).

Fig. 3: ML performance

operations and covers a wide range of domains. These benchmarks were chosen because they consist of several, error-prone floating-point instructions.

The evaluation of our ML model using the profiled workloads was completed under a computer configuration of 32-core AMD EPYC 7702 @2.00 GHz and 128GB memory.

4.2 Evaluating Considered ML Algorithms

We start by evaluating the performance of the four considered ML algorithms (see Section 3), in order to determine which method is the most appropriate for the task. Note that the majority of testing samples consists of erroneous ISQs, in order for each bit to have a representative amount of erroneous testing samples, otherwise the avg *Acc* value would be extremely misleading.

According to the results presented in Table 3a, **SVM** performs quite poorly. The low *Acc* and minimum *TPR* suggest that the model almost always predicts error-free bits. **RF** outperforms the rest of the algorithms in terms of *Acc* and *TPR*, and thereby better identifying the erroneous bits. Thus, we will focus on presenting the results achieved by **RF** in all following experiments.

Figure 3b shows the progress of the model when considering deeper instruction execution history, using the **RF** algorithm with default settings. Evidently, increasing the execution history from a depth of two to six, can have a significant positive impact on the performance of the model, confirming our hypothesis. We observe that *TPR* is already high and slightly increases. Most improvement is noted in *TNR* metric, indicating that the ability of our model to avoid mispredictions of non-erroneous bits as erroneous, is profoundly increased.

This improvement positively affects the *Acc* of the model. Further increase of the execution history (for $d > 6$) has minimal effect on the prediction metrics, and this can be

attributed to the fact that 6 stages is the depth of pipeline, thus $d = 6$ covers the possible in-flight instructions in the target FPU.

4.3 Cross-application Evaluation

We perform cross-application evaluation by testing our model ($d = 6$) under multiple unknown (to the classifier) applications and present detailed results about the performance. At this point, our RF-based model predicts simultaneously all 64 bits using as input an ISQ with depth $d = 6$. We use as reference the 64-bit model with $d = 2$, as it represents state-of-the-art approaches [5], [9].

Table 1 provides specific details regarding the performance of the models, using the *Acc*, *TPR* and *TNR* metrics. The performance varies depending on the application, which is anticipated, as each application consists of different ISQs and input data.

As shown in Table 1, the model that considers deeper instruction execution history ($d=6$) achieves higher performance across all metrics and applications, compared to the model when $d=2$. This is coherent and expected as per Figure 3b. More specifically, it achieves very high *Acc* and *TNR*, with an average value for all applications greater than 99%. This is expected, as the classes of the testing data are highly imbalanced, so the number of bits without error are much more abundant class than the number of bits showing timing errors.

Interestingly, the average *TPR* value for all applications is noticeably lower than the percentage shown in Figure 3b. However, such results are expected because the testing data used in Figure 3b, although unknown to the classifier, are extracted from the same applications used also for training, thus the instruction sequences might have similar patterns to those used in training. Nevertheless, the model with deeper instruction execution history achieves higher *TPR* values, indicating that the model predicts correctly more error-prone bits.

In general, the model that considers all in-flight instructions in the pipeline identifies more accurately error-prone bits. However, this improvement comes with a cost. In fact, the prediction complexity of **RF** is highly related to the number of trees (n_{trees}) of the 'forest' and the number of features (m) of the data (eq. 3), which is evident

Fig. 4: Significance-aware Code Vulnerability Factor (SCVF), Root Mean Squared Error (RMSE) and Application Vulnerability Factor (AVF) per application (under 25% delay increase and $d=6$).

TABLE 1: Accuracy and Inference speed of short history (d=2) model (state-of-the-art) and deep history (d=6) model & vulnerability evaluation time

history depth app	d=2			d=6			Time (mins)		
	avg Acc	avg TPR	avg TNR	avg ACC	avg TPR	avg TNR	RMSE	AVF	SCVF
HOTSPOT	95.42%	78.57%	95.40%	99.58%	82.55%	99.59%	86.9	3.38	3.35
KMEANS	100%	100%	100%	100%	100%	100%	50.2	2.03	2.01
SOBEL	96.93%	75.90%	96.92%	99.35%	80.39%	99.36%	28.5	1.14	1.13
SRAD	97.51%	76.20%	97.50%	99.72%	81.15%	99.73%	85.8	3.36	3.34
overall	97.46%	82.67%	97.45%	99.66%	86.02%	99.67%	-	-	-
inference time	0.06ms			0.14ms			-		

from the following equation:

$$RF_{P_{complexity}} = O(m \cdot n_{trees}) \quad (4)$$

The increased inference time of the improved model (2.33×) is utterly justified by the increased complexity of RF due to the additional considered features.

4.4 Analysis of Application Vulnerability to Errors

SCVF measures for each benchmark are displayed in Figure 4. Intuitively, we suggest that higher SCVF indicates higher output quality loss. To verify the proposed metric, we use as reference two metrics, the root mean squared error (RMSE) and the application vulnerability factor (AVF). RMSE, which is a commonly used metric [12], estimates the actual output quality loss incurred by the timing failures under 25% worst-case delay-increase for each application and is defined as follows:

$$RMSE = \sqrt{\frac{1}{\#ISQs} \cdot \sum_{n=0}^{\#ISQs} (O_{gold}(n) - O_{sim}(n))^2} \quad (5)$$

where $O_{gold}(n)$ and $O_{sim}(n)$ denote the error-free output and simulated output, respectively, obtained by post-layout gate-level simulation (PGS). Furthermore, we compare our approach with state-of-the-art approaches. We calculate the error application vulnerability factor (AVF) as proposed in existing works, using our ML-based predictions, where:

$$AVF = \frac{1}{\#ISQs} \cdot \sum_{n=0}^{\#ISQs} \frac{\#total\ erroneous\ bits}{\#total\ predicted\ bits} \quad (6)$$

Our evaluation includes applications with various output quality loss values, as shown by the RMSE in Figure 4 and Table 1. More specifically, in Kmeans do not occur any errors under 25% clock reduction, and thus, there is no quality loss, as indicated by all metrics. This proves that our metric is effective in applications that are robust in timing error manifestation under delay increase. Accordingly, SCVF reaches the highest value for the Hotspot program, which exhibits the highest quality loss amongst the tested benchmarks. Finally, the estimated SCVF value for Sobel and Srad properly reflects their quality loss. On the contrary, we observe that when using AVF, which is significance agnostic, there is no correlation between the estimated error vulnerability and the output quality loss, leading to inaccurate quality loss assessment. For instance, the calculated AVF value is very low in Hotspot application, although it has the highest output quality loss. Similarly, AVF fails to reflect the output quality loss of Sobel.

Additionally, as shown in Table 1, the calculation of SCVF is 25.94× faster than RMSE on average, which is a considerable acceleration.

5 CONCLUSIONS

In this paper, we propose a significance-aware application error vulnerability metric, which is estimated based on an instruction- and history-aware error prediction model. The developed ML model is considering simultaneously different types of instructions and all in-flight instructions in a pipeline. The ML model detects the exact location of erroneous bits with 99.66% average accuracy, outperforming the state-of-the-art. Furthermore, SCVF evaluates effectively the output quality loss in applications due to error manifestation, providing key insights, and the possibility of an early error impact evaluation.

REFERENCES

- [1] Y. Zhang *et al.*, "irazor: Current-based error detection and correction scheme for PVT variation in 40-nm ARM cortex-r4 processor," *JSSC*, vol. 5, pp. 619–631, 2018.
- [2] S. Borkar *et al.*, "Parameter variations and impact on circuits and microarchitecture," in *DAC*, 2003, pp. 338–342.
- [3] J. Constantin *et al.*, "Statistical fault injection for impact-evaluation of timing errors on application performance," in *DAC*, 2016, pp. 13:1–13:6.
- [4] C. Chang *et al.*, "Assessing the impact of timing errors on HPC applications," in *SC*, 2019, pp. 70:1–70:19.
- [5] X. Jiao *et al.*, "Clim: A cross-level workload-aware timing error prediction model for functional units," *IEEE Transactions on Computers*, pp. 771–783, 2018.
- [6] J. Constantin *et al.*, "Exploiting dynamic timing margins in microprocessors for frequency-over-scaling with instruction-based clock adjustment," in *DATE*, 2015, pp. 381–386.
- [7] B. Fang *et al.*, "epvf: An enhanced program vulnerability factor methodology for cross-layer resilience analysis," in *DSN*, 2016, pp. 168–179.
- [8] A. Azarpeyvand *et al.*, "Fast and accurate architectural vulnerability analysis for embedded processors using instruction vulnerability factor," *Microprocessors and Microsystems*, vol. 42, 2016.
- [9] X. Jiao *et al.*, "Tevot: Timing error modeling of functional units under dynamic voltage and temperature variations," *Dac*, 2020.
- [10] I. Tsiokanos *et al.*, "Defcon: Generating and detecting failure-prone instruction sequences via stochastic search," in *DATE*, 2020, pp. 1121–1126.
- [11] W. Liu *et al.*, "Design and analysis of inexact floating-point adders," *IEEE Trans. Computers*, pp. 308–314, 2016.
- [12] A. Botchkarev, "Performance metrics (error measures) in machine learning regression, forecasting and prognostics: Properties and typology," 09 2018.

Styliani Tompazi presently pursues her Ph.D. on machine learning based modelling of computing system reliability and power in the school of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, UK. She graduated from the University of Thessaly, Greece, in 2019, with M.Sc. in Electrical and Computer Engineering. Her field of interest is resilient and energy-efficient systems.

Ioannis Tsiokanos is currently a Research Fellow at the School of Electronics, Electrical Engineering and Computer Science of Queen's University Belfast. He received his Ph.D. in Electrical and Computer Engineering in 2021 from Queen's University Belfast. His research interests include energy-efficient and reliable systems, timing/power modelling, and hardware security.

Jesus Martinez del Rincon is presently a Senior Lecturer in the School of Electronics, Electrical Engineering and Computer Science of Queen's University Belfast, UK. He was awarded a Ph.D. in Computer Vision in 2008 from the University of Zaragoza. His research interests include deep/machine learning, computer vision and cybersecurity applications.

Georgios Karakonstantis is a Reader in the School of Electronics, Electrical Engineering and Computer Science of Queen's University Belfast, UK, and senior member of the IEEE. He received the M.Sc. and Ph.D. degree in Electrical and Computer Engineering from Purdue University, USA. His research focuses on energy-efficient and error-resilient computing and storage architectures.