DEVoT: Dynamic Delay Modeling of Functional Units Under Voltage and Temperature Variations

Dongning Ma, *Graduate Student Member, IEEE*, Xinqiao Zhang, *Graduate Student Member, IEEE*, Ke Huang, *Member, IEEE*, Yu Jiang, Wanli Chang, *Member, IEEE*, and Xun Jiao

***Abstract*—Timing errors of microelectronic circuits occur when the circuit timing specification is violated, i.e., the dynamic delay of circuits exceeds the circuit clock period. With the continuous scaling of CMOS technology, microelectronic circuits are increasingly susceptible to microelectronic variations such as variations in operating conditions. Such variations can cause delay uncertainty in microelectronic circuits, leading to *timing errors*. Circuit designers typically combat these errors using conservative guardbands in the circuit and architectural design, which can, however, cause significant loss of operational efficiency. In this article, we propose DEVoT, a supervised learning model that can predict the dynamic delay of functional units (FUs) under different operating conditions, clock speeds, and input workload. The main contribution of DEVoT is to jointly consider the impact of voltage, temperature, and input workload in path sensitization, hence predicting the dynamic delay. We measure the dynamic delay using switching activity generated through gate-level simulation of post place-and-route design in the TSMC 45-nm process. We characterize the delay of FUs under different operating conditions and input workload. We then extract useful features in the input workload that influences dynamic path sensitization. Using these features, we apply supervised learning methods to build DEVoT. Across 100 different operating conditions, four widely used FUs, and three datasets, DEVoT achieves, on average, less than 2% relative deviation from the ground truth and is 100× faster than the gate-level simulation. We present two case studies using DEVoT. First, we use DEVoT to predict timing errors of FUs, and DEVoT achieves an average prediction accuracy at 98.04%. We further use DEVoT to estimate application output quality under different operating conditions, and DEVoT achieves an average estimation accuracy at 97% for two image processing applications. Second, we present a fuzzing-based method to identify "critical" patterns that can cause longer delay for a given circuit. Built on top of DEVoT, the generated input patterns can improve the sensitized delay by up to 8.3% compared to random patterns. DEVoT also outperforms automatic test pattern generation (ATPG) in sensitizing circuit delay. We will open-source DEVoT, which can assist circuit designers to perform early design space exploration and can also help software developers in approximate computing community to assess their program resilience to hardware approximation without performing circuit simulation.**

***Index Terms*—Approximate computing, temperature variation, timing errors, voltage variation.**

I. Introduction

As THE transistor size scales down to deep nanometer era, microelectronic circuits are increasingly susceptible to microelectronic variability [21]. Generally speaking, microelectronic variability arises from various sources, such as operating conditions, manufacturing, or aging [21]. Due to the burgeoning use of microelectronic devices in mobile and wireless applications, the threat from variations in operating conditions, which is typically caused by supply voltage droops and temperature fluctuations, is continuously increasing. The most immediate manifestation of such variations is the delay uncertainty which can prevent circuits from meeting their timing specifications, resulting in *timing errors*. Without proper protection, such timing errors can pose great threats to the reliability of digital systems. Currently, circuit designers typically combat timing errors by adding safety margins to the voltage and/or the clock frequency, known as guardbands. However, this practice leads to overly conservative design as the margins often exceed 40% of the nominal target specifications [22].

To avoid such an pessimistic design while also protecting circuits from timing errors, designers must understand and model the timing effects, i.e., delay uncertainty or timing errors, of dynamic variations. Many models are proposed to predict the timing effects of arithmetic instructions [11], [37], [39] or functional units (FUs) [15], [38], [43], [44]. For example, instruction-level models [11], [37], [39] predict timing errors based on instruction types. B-hive [44] and HFG [38] both use machine learning methods to predict timing errors of FUs under dynamic variations.

Unfortunately, despite significant prior efforts, variation-induced timing effects still cannot be accurately modeled and, subsequently, exposed to the application level for a holistic evaluation. This is largely due to a lack of consideration of workload variation that can completely change the manifestation of dynamic variation in timing errors. Specifically, while variations in operating conditions can change the delay of (critical) paths and; hence, the static delay of a circuit, this delay is not the "real" circuit delay at a certain time. The

Manuscript received August 16, 2020; revised December 22, 2020 and March 16, 2021; accepted April 13, 2021. Date of publication April 30, 2021; date of current version March 21, 2022. This article was recommended by Associate Editor S. Reda. *(Corresponding author: Xun Jiao.)*

Dongning Ma and Xun Jiao are with the Department of Electrical and Computer Engineering, Villanova University, Villanova, PA 19085 USA (e-mail: dma2@villanova.edu; xun.jiao@villanova.edu).

Xinqiao Zhang and Ke Huang are with the Department of Electrical and Computer Engineering, San Diego State University, San Diego, CA 92182 USA (e-mail: xzhang10@sdsu.edu; khuang@sdsu.edu).

Yu Jiang is with the School of Software, Tsinghua University, Beijing 100084, China (e-mail: jy1989@mail.tsinghua.edu.cn).

Wanli Chang is with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China, and also with the Department of Computer Science, University of York, York YO10 5DD, U.K. (e-mail: wanli.chang.rts@gmail.com).

Digital Object Identifier 10.1109/TCAD.2021.3076970

"real" (dynamic) circuit delay is the delay of the sensitized longest path, and path sensitization behavior is actually determined by circuit workload (Section III). Actually, the same instruction or FU could exhibit a different delay under different input operands, resulting in different timing errors [27]. Unfortunately, due to the extremely large input space, incorporating input operands into timing delay/error modeling becomes very difficult, if not impossible.

As the efforts to reduce conservative timing margins become more aggressive, designers are opening up to continuing operations even in the presence of timing errors. Recently, the research community embraces *approximate computing* by allowing errors only in computation units that can tolerate them [15], [41], [43]. The output quality of approximate computing is hard to guarantee because it usually depends on the input data. Existing error models used in approximate computing include *single bit flip* [43], *bit flip with uniform probability* [15], and *last value* [41]. As a language for expressing approximate computation, Rely [6] allows developers to define a *reliability specification*, which identifies the minimum required probability with which a program must produce an exact result. By further enhancing the capabilities of Rely, Chisel [33] provides combined *reliability* and/or *accuracy* specification, which defines a maximum acceptable difference between the approximate and exact result values. Guaranteeing the former specification can be done through *unequal error protection* methods [3] or by carefully partitioning the computation through reliable or unreliable media [10]. However, meeting the latter specification remains a challenge for automatic model generation, since the model must provide reliable information about the possibility of an error occurrence under different workload conditions, i.e., accurate *error prediction*.

To address the aforementioned challenges, we propose **DEVoT**, a dynamic delay model of FUs by jointly considering workload variations and variations in operating conditions based on [51]. The key idea of **DEVoT** is to establish a prediction model that can best explore the relationship from input features to sensitized circuit paths by learning the existing patterns and their sensitized delay. **DEVoT** can predict dynamic delay of FU for a given input data, voltage, and temperature condition. First, under a wide variation of workload and operating conditions, we characterize the dynamic delay behavior of FUs by analyzing switching activity file using gate-level simulation of post-layout designs in TSMC 45-nm technology. Based on such characterization, we extract the useful features. We then apply supervised learning methods to fit the extracted features and the corresponding dynamic delay, which leads to **DEVoT**. Note that such a training process is a one-time effort. Next, we use **DEVoT** to predict the dynamic delay of FUs and compare it with simulation-based ground truth so as to evaluate the prediction accuracy. The advantages of **DEVoT** are: 1) the prediction of delay can be reused across multiple clock periods without developing error models for each clock period [25], [26]; 2) improved accuracy by considering input data; 3) accelerated execution over gate-level simulation so that it can assist circuit designers to perform early design space exploration; and 4) friendly interface for software developers to use in approximate computing as **DEVoT** is a standalone Python module to be opensourced for the research community. Specifically, our contributions are as follows.

1) We perform dynamic timing analysis (DTA) to characterize circuit dynamic delay under a wide range of workload and operating conditions, based on which we collect training data. The DTA is based on the analysis of switching activity generated from extensive gate-level simulations with timing information extracted from a standard ASIC flow that considers physical details of post-layout designs in TSMC 45 nm.
2) We extract useful features from the training data and apply supervised learning methods to build **DEVoT** that can predict dynamic delay of FUs under different input workload, voltage, and temperature conditions. To the best of our knowledge, **DEVoT** is the first delay model that can jointly consider the workload variations and variations in operating conditions. We apply and evaluate various learning methods and finally select random forest (RF) as the modeling method.
3) We evaluate the performance of **DEVoT** by comparing the predicted delay with the simulation-based delay measurement. **DEVoT** achieves on average less than 2% relative deviation and is 100× faster than the gate-level simulation across 100 different operating conditions, five widely used types of FUs, and three datasets.
4) We use **DEVoT** to predict the timing errors of FUs across 100 different operating conditions, five widely used types of FUs, three clocking speeds, and three datasets. The results show that **DEVoT** achieves an average prediction accuracy of 98% in predicting timing errors.
5) We also expose circuit-level errors to application level for a holistic evaluation, based on which **DEVoT** can estimate application quality under different operating conditions. The results show that **DEVoT** achieves an average prediction accuracy of 97% in estimating the output quality of two widely used image processing applications: a) the Sobel filter and b) the Gaussian filter.
6) We develop **DFuzz**, a fuzzing-based method that aims to identify "critical" input sensitizing long paths for a given circuit, without requiring any netlist topology information or preliminary path analysis.
7) We propose a delay-guided method to heuristically search for critical input patterns by selectively preserve "promising" seeds for the iterative fuzzing process. We develop various mutation strategies, including deterministic and nondeterministic to maximize the possibility of generating critical patterns. We also develop parallel fuzzing to further optimize the efficacy of **DFuzz**.
8) We evaluate **DFuzz** on five widely used types of FUs. The results show that **DFuzz**-generated input patterns can improve the delay measurement based on random patterns by up to 8.3% and automatic test pattern generation (ATPG) by up to 21.23%.

The remainder of this article is organized as follows. Section II presents the related work. Section III formulates

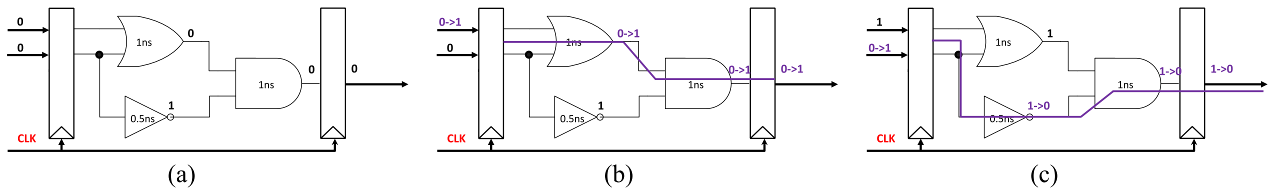


Fig. 1. Different delay under different inputs. (a) Initial state. (b) First input changes (delay = 2). (c) Second input changes (delay = 1.5 ns).

the modeling problem and defines useful terms. Section IV describes the process of constructing and evaluating **DEVoT**, including dynamic delay characterization, input feature extraction, and the application of supervised learning methods. Section V presents the use case of **DEVoT** in predicting timing errors and application quality. Section VI presents the use case of **DEVoT** in generating critical input patterns. We conclude our work in Section VIII.

II. Related Work

To combat microelectronic variability, researchers proposed better-than-worst-case (BTWC) design methods to enable guardband reduction. A shadow flip-flop was used in [14] to detect and correct any timing errors induced by speculated voltage scaling. Such shadow flip-flop approaches were also used in error-detection sequential circuit (EDS) [4] to double the sample and compare the signal at different timing. However, the overhead with such intrusive error detection and correction techniques can be large, e.g., 18% [34] and 21% [17] overheads in area, and 8% [7] power overhead.

A less intrusive way to combat variability is to model the delay in advance and then adaptively change the clock speed to improve efficiency. Various models are proposed to predict dynamic delay or timing errors of arithmetic instructions [11], [37], [39] or FUs [15], [38], [43], [44]. Instruction-level models predict timing errors based on the maximum delay of each instruction measured during simulation [11], [37], [39]. This method considers only instruction type to discriminate between instructions. For example, Rahimi *et al.* used a large-scale gate-level simulation to identify critical instructions [37] and critical instruction sequences [39] based on the timing error rates (TERs) caused by them. Constantin *et al.* also used a large-scale simulation to measure the maximum delay that can be caused by each instruction type and use the maximum delay as the basis to perform online dynamic frequency scaling. All these works are based on simulation with representative workload profiled from benchmarks.

Going down to the circuit level, B-Hive [44] divides timing errors into five categories and classifies them using decision trees; HFG [38] uses linear discriminant analysis to predict variability-induced errors of FUs. MACACO [46] uses the Monte-Carlo simulation to model timing errors of voltage-scaled adders and multipliers. VARIUS [42] uses probabilistic analysis to model timing errors of microarchitectural blocks. Error models of voltage-scaled FUs are also intensively used in approximate computing research to assess the application-level effects. For example, *single bit flip* model flips one randomly chosen bit [43]; *bit flip with uniform probability* model flips different bits uniformly with a per-FU error probability [15], [28]. However, none of these works considers the impact of workload variations in predicting timing errors.

Recent research [25], [26] suggests the significant impact of input workload in timing error profile. These studies model the timing errors using machine learning methods with the consideration of the input workload. However, **DEVoT** is different from existing studies [25], [26] in several aspects. First, existing studies focus on modeling timing errors while **DEVoT** focuses on delay prediction. The advantage of delay prediction is that the predicted delay can be reused across any given clock periods to determine timing errors, while the existing studies need to develop error models for each clock period, significantly limiting its scalability. Second, **DEVoT** considers a holistic dynamic variations, including both voltage and temperature with 100 different operating conditions while the existing studies only consider clock variations [25] or voltage variations [26]. Third, the usage of **DEVoT** is beyond the timing error prediction and estimation of approximate computing. As shown in Section VI, **DEVoT** is central to the development of **DFuzz** in DTA.

III. Problem Formulation

Timing errors only occur when the clock period does not meet the circuit delay [9]. Circuits have two types of delays: 1) static delay and 2) dynamic delay. Static delay refers to the delay of the critical path in the circuit, which can be affected by variations of operating conditions (we focus on dynamic variations here and but the same principle can be used to incorporate process and aging variations). This delay, however, is not useful in timing error prediction because the critical path may not get sensitized. Actually, the critical path is rarely sensitized by real-world workload [11]. In order to predict timing errors, we need to compare circuit clock period with the sensitized dynamic delay. The dynamic delay is the delay of the sensitized longest path in the circuit, which is determined by the input workload. For example, as illustrated in Fig. 1, the dynamic delay in Fig. 1(b) is 2 ns while the delay in Fig. 1(c) is 1.5 ns. This is because different circuit paths get sensitized as marked. Thus, we represent the dynamic delay as a function f_d of input workload and operating conditions, as shown in the following:

$$D = f_d(V, T, I) \tag{1}$$

where V and T represent respective voltage and temperature condition, I is the input workload to the circuit, and D is the dynamic delay of an FU. Once a delay is predicted, such delay can be reused to predict timing errors across different clock speeds by comparing the delay with clock periods.

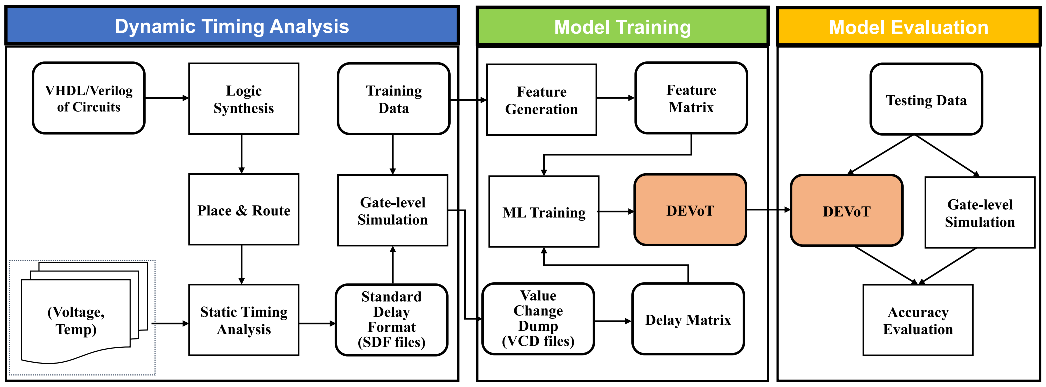


Fig. 2. Construction and evaluation of **DEVoT**.

Our goal is to learn (an approximation of) f_d given a set of inputs and operating conditions without any knowledge of the circuit structure. However, the potential input space of workload is huge. For a circuit with two 32-bit inputs, the potential input space is 2^{64}. Therefore, we propose to evaluate a set of supervised learning methods to classify the inputs.

TABLE I
OPERATING CONDITION PARAMETERS

	Start Point	End Point	Step	of Points
Voltage	0.81V	1.00V	0.01V	20
Temperature	0°	100°	25°	5
Clock Speedups	5%	15%	5%	3

IV. **DEVoT** MODEL

DEVoT is comprised of three phases as shown in Fig. 2: 1) DTA; 2) *model training*; and 3) *model evaluation*.

1) The DTA phase implements the standard ASIC flow and uses gate-level simulation to generate dynamic delay under different input workload and operating conditions.
2) In the *model training* phase, we extract useful features from training data and apply supervised learning methods to train **DEVoT**.
3) In the *model evaluation* phase, **DEVoT** predicts dynamic delay across different FUs, datasets, and operating conditions, which are compared with the simulation-based ground truth to evaluate prediction accuracy.

More details about the three phases are illustrated as follows.

A. *Dynamic Timing Analysis*

The purpose of the DTA phase is to generate dynamic delay under different workload and operating conditions. We focus on four FUs, 32-bit integer adder (INT_ADD) and multiplier (INT_MUL), and 32-bit floating point adder (FP_ADD) and multiplier (FP_MUL). These four most-widely used FUs are basic computation blocks for applications, such as image-processing and deep learning applications and have been main modeling targets of similar works [38], [44]. The floating-point units (FPUs) are compatible with the IEEE-754 standard and can provide more complex circuit structures compared to their integer counterparts.

To collect the training input data, we generate the random input data as stimuli for simulation. For a 32-bit vector, we randomly set each bit independently with a homogeneous distribution to produce the training data. But note that testing input data might come from application profiling.

We perform logic synthesis and place&route to generate gate-level netlists. Then, we perform static timing analysis (STA) to generate standard delay format (SDF) files under different operating conditions. To inject the dynamic variations, i.e., different voltage and temperature conditions, we use voltage–temperature scaling features of EDA tools to enable the composite current source method for modeling cell behavior. Thus, we generate an SDF file for each voltage–temperature (V, T) pair. We use 20 different voltage points and five different temperature points as shown in Table I [in total 100 different (V, T) pairs].

We then feed SDF files to gate-level simulation to perform back-annotation simulation. Since the main purpose of gate-level simulation is to generate the value change dump (VCD) file for DTA, we perform simulation with a relatively slow clock period to make sure there are no timing errors. VCD files contain the switching activity of interested circuit nodes, e.g., output bits, in the circuit. With such detailed switching activities, we then compute the dynamic delay at each cycle. To get a dynamic delay at some cycle N, we use the time of the very last toggled event at the input pins of all sequential elements t' to subtract the arrival time of the positive clock edge t. For example, t' could be the time of the last toggled output bit. That is, the dynamic delay at this cycle is $t' - t$. We develop a Python script that can automatically parse VCD files to extract dynamic delay at each cycle.

B. *Model Training*

1) Feature Generation: The purpose of *feature generation* is to extract "useful variability feature" from raw training data. As presented in (1), dynamic delay is a function of operating conditions V, T, and workload I. Operating conditions are typically at limited discrete levels such as shown in Table I; hence, it is possible to train the model using all operating conditions. However, the potential input space of the workload is huge. For a circuit with two 32-bit inputs, the potential input space is 2^{64}. It is not feasible to apply all 2^{64} input patterns for training. Therefore, the key accomplishment of **DEVoT** is to predict timing errors under unseen input data by learning the path sensitization behavior under unseen input data.

We convert all input data to bit-level vectors since circuits receive binary input; since each bit affects path sensitization, each bit value is an individual feature. Next, we need to determine if we need to incorporate history inputs. To do this, we perform a path sensitization analysis. According to [5], a sensitized path would have all of its nodes toggled. For a node to be toggled, the current signal value at the node needs to be different than the previous one. Thus, for a combinational circuit as shown in Fig. 1, both the previous and current input have impact on the path sensitization. For example, when the second input changes, whether the value of a node toggled depends on its current state, which is set by the previous input. Thus, the previous input sets the state and current input toggles the circuit nodes based on current state. Thus, we include current input $x[t]$ (concatenation of multiple inputs), and history input $x[t-1]$ as our feature. We verify this conclusion by performing the following two simulation scenarios.

1) *Scenario 1:* We fix the current input $x[t]$ and randomly vary the preceding cycle input $x[t-1]$, where we set cycle $t = 10, 30, 50, 70, \ldots$ We use this to evaluate the effects of immediately preceding input.
2) *Scenario 2:* We fix both the current input $x[t]$ and the immediately preceding input $x[t-1]$, while randomly varying the preceding input of immediately preceding input $x[t-2]$, where we set t as above. We use this to evaluate the effects of the deeper history.

We observe that in Scenario 1, $D[t]$ varies irregularly. In Scenario 2, $D[t]$ is also fixed.

Therefore, our variability feature is $\{V, T, x[t], x[t-1]\}$ and output label is $D[t]$, which can then be used to predict timing errors across different clock periods. After *Feature Generation*, we can generate the following feature matrix I and delay matrix D, where $x[t] \in \{0,1\}^K$, and V and T are real numbers. For 32-bit circuit, $K = 64$ because $x[t]$ and $x[t-1]$ each has 64 bits. This makes a feature dimension of 130 and the possible input feature space is more than 2^{130}

$$I = \begin{bmatrix} x[t] & x[t-1] & V^1 & T^1 \\ x[t+1] & x[t] & V^2 & T^2 \\ \vdots & \vdots & \vdots & \vdots \\ x[t+N] & x[t+N-1] & V^N & T^N \end{bmatrix} D = \begin{bmatrix} D[t] \\ D[t+1] \\ \vdots \\ D[t+N] \end{bmatrix}. \tag{2}$$

2) ML Training: While specific classes of circuits show certain positive learnability results [32], they do not cover the circuits we consider here. In contrast, we focus on learning when a circuit does not work as desired, i.e., the circuit contains timing errors. Capturing the timing errors requires learning the dynamic path sensitization by specific input workload. Thus, we evaluate several widely used supervised learning classification methods: k-nearest neighbor (k-NN), support vector machine (SVM), linear regression (LR), and RF for their increased sophistication and practical use.

k-NN provides useful theoretical properties [12] and has limited parameters to train. k-NN predicts the target by local interpolation of the targets associated of the K nearest neighbors in the training set. However, in our case, k-NN finds its nearest neighbors based on the Hamming distance between input vectors, which actually overlooks the situation wherein different bit positions would have disparity of significance on path sensitization. LR and SVM can learn weights w on each feature including each bit position. By using these two methods, we consider the disparity of significance of different bit positions in sensitizing paths. However, the major limitation of these two methods is that they put a fixed weight on each bit position. This is not suitable here because each bit position does not contribute linearly to the final timing error, and the contribution of each bit position might be changed along with the change of other bit values. For example, for an "AND" gate—if one input is 0, then the final result will always be 0 regardless of the other input. To address this problem, we propose to use RFC. RF is an ensemble learning method that constructs multiple decision trees and uses majority votes to improve accuracy and prevent overfitting. Decision trees are a nonparametric supervised learning method that aims to establish a set of decision rules from training data. This method emphasizes the disparity of different bit positions and more importantly, it also considers the interaction between different bit positions. This is consistent with the circuit path sensitization because the sensitization is potentially determined by the variance of all bit positions.

TABLE II
PREDICTION ACCURACY, TRAINING, AND TESTING TIME

method	MAPE	Training Time	Testing Time
LR	13.4%	6.18s	2.1s
KNN	15.1%	125s	3648s
SVM	7.2%	15749s	9997s
RFC	1.2%	141s	3.3s

Table II presents the prediction accuracy [mean absolute percentage error (MAPE) as explained next], training and testing time of four methods using 200K training data and 200K test data under a computer configuration of 2-core Intel Xeon CPU E5504@2.00 GHz and 50-GB memory. Based on the results, we choose RF due to its high accuracy, fast computing time and superior interpretability. Actually, RF fits our task scenario better than other methods because it can interpret the significance disparity between different features (compared with KNN) and it considers the interactions among different bits/features (compared with SVM and LR). Since the training process is a one-shot offline activity, the testing time is more important for users. We will opensource the pretrained models which are stand-alone Python modules for research community.

C. Evaluation

1) Evaluation Metrics: For a given input workload, our model will predict the delay caused by it. We use the widely used MAPE as our evaluation metric, as shown in the following:

$$\text{MAPE} = \frac{|\text{Delay}_{\text{pred}} - \text{Delay}_{\text{real}}|}{\text{Delay}_{\text{real}}}. \tag{3}$$

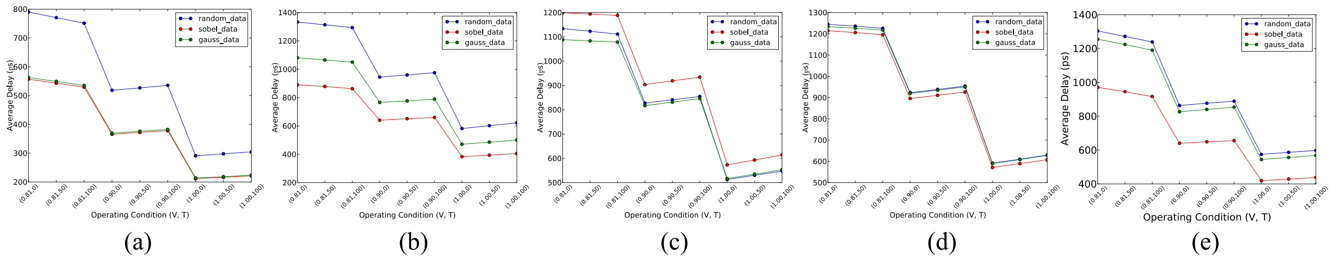


Fig. 3. Average delay under different datasets and operating conditions. (a) INT_ADD delay. (b) INT_MUL delay. (c) FP_ADD delay. (d) FP_MUL delay. (e) INT_MAC delay.

TABLE III
TIMING ERROR PREDICTION OF DIFFERENT LEARNING METHODS

method	Accuracy	Training Time	Testing Time
LR	82.3%	6.84s	2.24s
KNN	81.7%	127s	3548s
SVM	92.2%	15653s	9879s
RFC	98.3%	142s	3.5s

We compare our model against the following baseline model that can help us evaluate the true performance of our model.

1) *Baseline:* This model is widely used in predicting the delay (and timing violations) of a specific unit or instruction [11], [37], [38], where the delay of an FU is modeled as the maximum delay sensitized by a set of representative inputs using massive simulation. For example, in [11], a large-scale simulation is used to simulate representative workload from hand-written kernels as well as semirandom test-cases, and use the maximum measured delay as the delay for instruction. Therefore, each instruction is predicted to have a fixed delay.

2) Setup: We generate the RTL descriptions of FUs using FloPoCo [13]. We perform logic synthesis, place&route, and STA using the Synopsys Design Compiler, IC Compiler, and PrimeTime, respectively, with TSMC 45-nm technology. We perform gate-level simulation using Mentor Graphics ModelSim. We use three datasets, random dataset, and real-world datasets profiled from two image processing applications from AMD APP SDK v2.5 [1], the Sobel filter, and Gaussian filter. The images are from the butterfly image dataset in Caltech-101 [16]. For training, we use 200K randomly generated data and 5% randomly picked images as training data; for testing, we use 200K (unseen) random data and the rest images as testing data. We profile application datasets by simulating the OpenCL codes of these applications with a customized *Multi2Sim* [45] simulator, a cycle-accurate CPU-GPU heterogeneous architectural simulator. We also perform error injection, i.e., inject timing errors back to applications, using *Multi2Sim* to obtain application-level quality. We use 100 operating conditions as shown in Table I. We adopt machine learning methods from Scikit-learn [35] with default hyperparameters (e.g., ten trees, all features considered during split for RF).

3) Dynamic Delay Characterization: We first characterize the dynamic delay variations under different operating

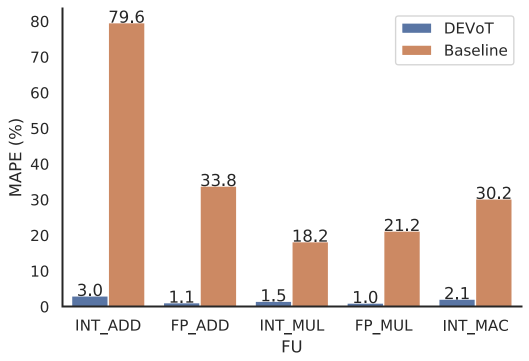


Fig. 4. Per-FU **DEVoT** accuracy.

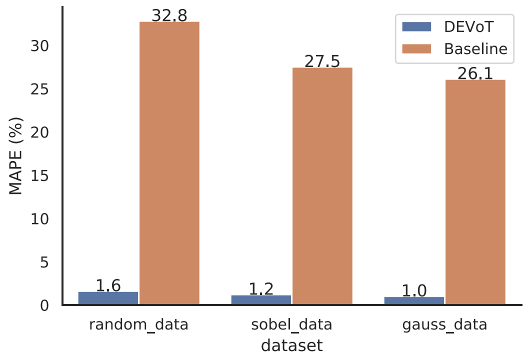


Fig. 5. Per-dataset **DEVoT** accuracy.

conditions and datasets. Specifically, for each dataset and operating condition, we compute the average dynamic delay across the entire dataset. Fig. 3 presents an example of 9 (V, T) pairs ($V \in \{0.81, 0.90, 1.00\}$ and $T \in \{0, 50, 100\}$) and three datasets, based on which we can observe several important facts. First, the dynamic delay is varied with different operating conditions. Specifically, as voltage increases, the delay is reduced. However, the impact of temperature on delay is not fixed. For example, when the voltage is 0.81 V, the increased temperature reduces delay; when the voltage is 0.90 V, the increased temperature increases the delay. This phenomenon is known as the inverse temperature dependence [47]. Second, the delay can also be changed dramatically by different datasets. For example, for INT_ADD, the average delay of the random dataset is 30% greater than that of application datasets. This indicates the significant impact of input workload on dynamic delay, thus motivating our consideration of the workload variations in delay modeling.

TABLE IV
AVERAGE TIMING ERROR PREDICTION ACCURACY OF **DEVoT** ACROSS 100 OPERATING CONDITIONS AND THREE CLOCK SPEEDS

FU	random data				sobel data				gauss data			
	DEVoT	**Delay-based**	**TER-based**	**DEVoT**-NH	**DEVoT**	**Delay-based**	**TER-based**	**DEVoT**-NH	**DEVoT**	**Delay-based**	**TER-based**	**DEVoT**-NH
INT_ADD	99.9%	0.02%	96.5%	91.9%	99.2%	0.77%	82.0%	86.7%	99.7%	0.01%	65.7%	82.5%
FP_ADD	98.6%	7.6%	87.1%	89.9%	98.2%	16.3%	80.2%	83.4%	98.6%	3.2%	78.1%	91.5%
INT_MUL	97.1%	21.1%	73.7%	83.7%	99.2%	0.39%	17.7%	31.4%	99.5%	6.2%	79.8%	69.7%
FP_MUL	97.0%	10.2%	84.9%	87.5%	95.4%	9.4%	73.1%	80.1%	96.7%	11.4%	82.1%	85.4%
INT_MAC	96.6%	29.2%	74.6%	82.5%	95.2%	11.5%	78.0%	85.6%	99.9%	21.9%	72.7%	95.4%

4) Delay Prediction Accuracy: In this section, we describe the performance of **DEVoT**. Fig. 4 presents the average per-unit prediction accuracy of **DEVoT** and Baseline for five FUs across three testing datasets. **DEVoT** exhibits an per-unit MAPE ranging between 1.1% and 3.0% and achieves an average per-unit MAPE 1.8% over all units. The Baseline model achieves an average per-unit MAPE at 36.6%. Thus, compared to the Baseline model, **DEVoT** exhibits 20.3× higher per-unit prediction accuracy.

Similarly, we also present the average per-dataset prediction accuracy of **DEVoT** and Baseline for three datasets across five FUs. As shown in Fig. 5, **DEVoT** exhibits an per-dataset MAPE ranging between 1.0%–1.6% and achieves an average per-dataset MAPE 1.3% across all datasets. The Baseline model achieves an average per-dataset MAPE at 28.8%. Thus, compared to the Baseline model, **DEVoT** exhibits 22× higher per-dataset prediction accuracy.

Furthermore, **DEVoT** is 100× faster than gate-level simulation on average across different FUs. In particular, **DEVoT** achieves even greater acceleration over simulation on more complex the circuit structure, e.g., floating-point units. This is because the more complex the circuit, the slower the simulation. But for **DEVoT**, because it is based on a fixed set of decision rules, it will not scale up with the complexity of the circuit.

V. TIMING ERROR PREDICTION

This section presents a case study of using **DEVoT** to predict timing errors of FUs across 100 operating conditions, three datasets, and four FUs. For each operating condition, we use three clock speedups (5%, 10%, and 15%) from its fastest error-free clock frequency (so that the output has timing errors). For a given input I, voltage V, temperature T, and clock speed t_{clk}, **DEVoT** classifies the corresponding output as either *correct* or *erroneous*. We evaluate the model performance using prediction accuracy and compare it with baseline models. The prediction accuracy is obtained by comparing **DEVoT** predicted result with simulation results

$$\text{prediction\_accuracy} = \frac{\#\text{matched\_cycles}}{\#\text{total\_cycles}} \tag{4}$$

where #total_cycles is the number of total simulation cycles and #matched_cycles is the number of cycles at which the predicted result matched the simulation result, i.e., both results are either C_c or C_e.

We compare **DEVoT** against the following baseline models that can help us evaluate the true performance of **DEVoT**.

1) *Delay Based:* This model is from [11], [37], and [38] where a timing error is predicted if the clock period does not meet the maximum delay measured offline at each operating condition. This model does not consider input workload but only instruction types, V, and T.
2) *TER Based:* This model is from [15] and [41], where a timing error is predicted with a probability based on the TER measured during offline simulation. This model is widely used in approximate computing.
3) ***DEVoT**-NH:* This model is trained similarly to **DEVoT** except it does not consider computation history, i.e., preceding input $x[t-1]$, as input features.

A. Timing Error Prediction

Table III presents the prediction accuracy, training and testing time of four learning methods using 200K training data and 200K test data under a computer configuration of 2-core Intel Xeon CPU E5504@2.00 GHz and 50-GB memory. The observation also verifies the superiority of RF in predicting timing effects. Table IV presents **DEVoT** prediction accuracy against the baseline models: Delay-based, TER-based, and **DEVoT**-NH. We compute the average prediction accuracy of such models across 100 operating conditions and three clock speeds as shown in Table I. We can observe several important facts. First, for all the datasets and FUs, **DEVoT** can exhibit prediction accuracy beyond 95%. On average, **DEVoT** exhibits 98.04% prediction accuracy. As a comparison, Delay-based exhibits extremely low prediction accuracy (9.94% on average) because of its pessimistic prediction: it always predicts timing error when the clock period does not meet its measured maximum delay. This means that whenever there is a clock speed up, it predicts timing errors, ignoring the case that the input workload may sensitize smaller delay. The TER-based model achieves, on average, 75.08% accuracy. The TER-based model uses a predetermined error probability from training data to predict testing data, without using any information from testing data. However, the delay/error statistics of training data and testing data may deviate significantly. Finally, **DEVoT**-NH presents an average accuracy of 81.8%. The difference between **DEVoT** and **DEVoT**-NH indicates the importance of incorporating the history input workload. Furthermore, **DEVoT** is 100× faster than gate-level simulation on average across different FUs. Typically, the more complex the circuit structure, the slower the simulation. But for **DEVoT**, because it is based on a fixed set of decision rules, it will not scale up with the complexity of the circuit.

TABLE V
APPLICATION QUALITY ESTIMATION ACCURACY USING FOUR MODELS

Application	**DEVoT**	**Delay-based**	**TER-based**	**DEVoT**-NH
Sobel	97.6%	75.7%	53.8%	58.8%
Gauss	96.5%	84.1%	64.6%	71.2%

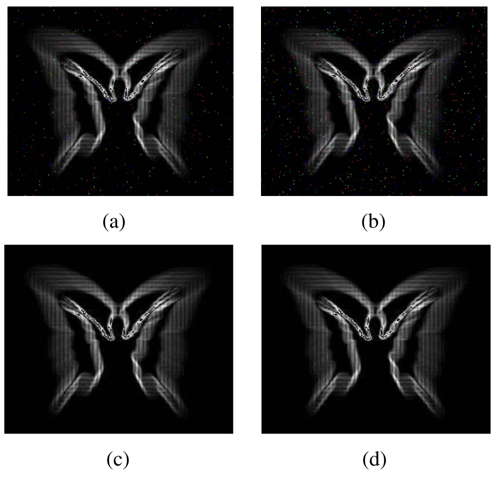


Fig. 6. Sobel filter output based on simulation (ground truth), **DEVoT**, **DEVoT**-NH, and TER-based models (the noisy pixels are more visible on electronic version or color printing). Note that we do not put delay-based model here because it always leads to completely corrupted output. (a) Ground truth (27 dB). (b) **DEVoT** (25 dB). (c) **DEVoT**-NH (56 dB). (d) TER-based (48 dB).

B. Application Quality Estimation

We present a case study of using **DEVoT** to estimate the application output quality under different operating conditions. Specifically, for each output image of the Sobel filter and Gaussian filter, **DEVoT** can classify it into either *acceptable* (PSNR $\geq$30dB) or *unacceptable*. This is especially important in approximate computing for exploring quality-energy tradeoff.

At each operating condition and clock speed, we use gate-level simulation, **DEVoT**, Delay-based, TER-based, and **DEVoT**-NH to derive the corresponding TERs of FUs. Then, we inject timing errors based on these TERs to applications using the *Multi2Sim* simulator. During the error injection process, we let the FUs return a random value each time they have timing errors, similar to [24]

$$\text{estimation\_accuracy} = \frac{\#\text{matched\_estimations}}{\#\text{total\_estimations}}. \quad (5)$$

We use (5) to compute the estimation accuracy. As shown in Table V, **DEVoT** presents on average 97% estimation accuracy, while the baseline models present 79.9%, 59.1%, and 65% average accuracy. Specifically, Delay-based would always estimate the output quality as *unacceptable* because it always predicts timing errors when there is clock speedup. This prediction may be consistent with actual outputs. For example, Fig. 6 shows an *unacceptable* output (27 dB) of the Sobel filter: **DEVoT** estimates correctly because its output is 25 dB; TER-based and **DEVoT**-NH are incorrect because their estimations are *acceptable*.

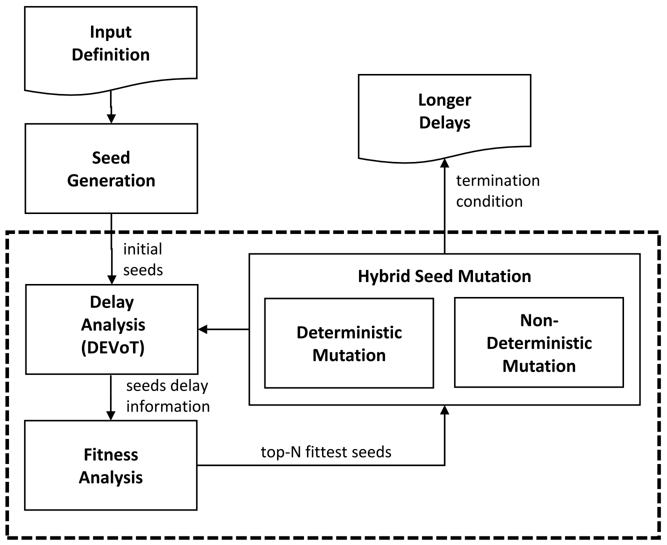


Fig. 7. **DFuzz** overview.

VI. CRITICAL PATTERN GENERATION

This section presents a case study of using **DEVoT** to generate critical patterns of FUs. Test pattern generation is critical to the success of DTA/verification. As opposed to STA, which is known to result in pessimistic timing margin based on the critical path computed from a multicorner worst case analysis at design time [11], DTA is widely deployed because it relies on a simulation-based method that can eliminate the false path problem. The quality of DTA heavily depends on the quality of test patterns. While ATPG methods has been widely used to generate high-quality test patterns, they typically require a preliminary path analysis to select the critical path, and then generate specific test patterns to sensitize this critical path [8], [18], [29], [31]. Compared to ATPG methods, we present a case study of using **DEVoT** to generate test patterns without preliminary path analysis and topology knowledge based on fuzz testing methods. Fuzz testing is broadly used to locate and identify software vulnerabilities or bugs [2], [19], [20], [36]. Recently, fuzz testing has been deployed in circuit functional verification [30]. Based on the *American Fuzzy Lop* (AFL) [49], RFuzz [30] mutates test patterns and achieve considerable coverage improvements compared to random testing. This article presents the first application of fuzzing method in timing analysis.

Fig. 7 presents the overview of **DFuzz**. First, we generate initial qualified input patterns based on input definition, which are known as "seeds" in fuzzing practice and are the basis for subsequent mutation process. Then, we analyze the circuit delay caused by each test pattern, based on which we analyze the fitness of each test pattern. This fitness determines what seeds can survive for future mutations. Then, we will perform mutation on these surviving seeds to generate new test patterns. The mutation strategy includes both deterministic and nondeterministic mutations. This process will be iterated until the termination condition is met. We describe the fuzzing process in detail as follows.

A. Input Definition and Seed Generation

In order to generate initial qualified seeds, we need to define the format of a test pattern first. In software testing [2], [19], [20], defining input is straightforward—the input is simply the input required for each program execution. For example, for image processing applications, the input is an image. However, input definition in DTA is nontrivial. As shown in Fig. 1, the delay of a circuit is actually determined by both current and previous input. Therefore, we define the input as a combination of previous input and current input, i.e., $\{x[t], x[t-1]\}$, where $x[t]$ is the current input and $x[t-1]$ is the previous input. Note that $x[t]$ is a combination of multiple inputs for a circuit. Eventually, the fuzzing input is a $4 * N$-bit vector, where N is the number of bits per input port. For example, for a 32-bit circuit with two inputs, the fuzzing input has 128 bits.

TABLE VI
DETERMINISTIC FUZZING STRATEGIES IN **DFuzz**

Name	Description
seq bit flip	sequentially flip each bit
seq byte flip	sequentially flip each byte
arith	perform arithmetic with given number on a byte
replace	replace a byte with given number
shift	circular shift for one bit

TABLE VII
NONDETERMINISTIC FUZZING STRATEGIES IN **DFuzz**

Name	Description
rdm bit flip	randomly flip a bit
rdm byte flip	randomly flip a byte
rdm arith	perform arithmetic with random number on a byte
rdm replace	replace a byte with random number
rdm shift	circular shift for random bit(s)

B. Delay Analysis

In software testing, the fuzzing process is guided by the coverage, such as branch coverage, resource utilization, etc. Different from these studies, **DFuzz** aims to generate test patterns that can lead to longer delay. Thus, we define the delay as our "coverage", enabling a delay-guided fuzzing process.

The typical process of measuring circuit delay caused by an input requires gate-level simulation via commercial simulators, such as ModelSim. However, it is infeasible to use simulation here to provide delay measurement. This is because during fuzzing process, we need to measure the delay sensitized by *each input* and use it as feedback. For simulation, this means that for each input, ModelSim command needs to be called, and the calling the ModelSim command alone is very time consuming. Then, the ModelSim needs to generate VCD files, and then we need to call another script to parse the VCD file to calculate the delay. This involves intensive context switching and file I/O in operating system and is not feasible for iterative mutation with real-time requirement during the fuzzing process.

Due to the friendly interface of **DEVoT**, which is essentially a standalone Python module, all we need to do is to directly feed the input to **DEVoT** and the input-specific delay can be automatically obtained with high accuracy.

C. Fitness Analysis

After measuring the delay caused by each test pattern, we need to decide which pattern(s) are "interesting" and can survive to generate child patterns. This process is performed using a fitness function. In **DFuzz**, we implement different fitness functions.

1) *Single Maximum:* This means that we only retain the pattern that causes the maximum delay at each generation and add it to the seed pool for future mutation. The maximum delay is based on the comparison of delay sensitized by all inputs in the current pool.
2) *Average Maximum:* This means that we retain the pattern that causes the maximum delay across n generations and add it to the seed pool for future mutation. Compared to the first function, this one considers a longer term effects of a pattern.
3) *n-Maximum:* This means that we retain n patterns that cause the top n maximum delays. Compared to the first function, this one considers multiple patterns that may allow for a more diverse seed selection.

Based on these fitness functions, the top-N fittest seeds will be used for the hybrid seed mutation while the rest seeds will be eliminated.

D. Hybrid Seed Mutation

We develop hybrid mutation strategies as shown in Tables VI and VII, which are classified into two categories: 1) deterministic and 2) nondeterministic.

Deterministic: Deterministic strategies starts with the intuitive ones—sequential bit and byte flips to more sophisticated ones, such as arithmetic (add or subtract a specific number) and replacement.

Nondeterministic: Deterministic strategies may saturate quickly before reaching the active termination condition. Therefore, we also develop nondeterministic strategies, which, instead of having a fixed mutation routine, have more randomness towards mutating seeds.

After the hybrid seed mutation, **DFuzz** will send the new child seeds to delay analysis and fitness analysis again for a new cycle of mutation in the fuzzing process.

E. Termination Condition

The fuzzing process iterates until a termination condition is met. We define two types of termination conditions—1) active and 2) passive.

Active Termination: The active termination condition is usually related to the limitation of resources or requirement of performance. For example, a user may set the limitation of running time. To balance the fuzzing performance and efficiency, we define three types of active termination conditions that can be customized by users: 1) running time T; 2) generation cycles i; and 3) sensitized delay d.

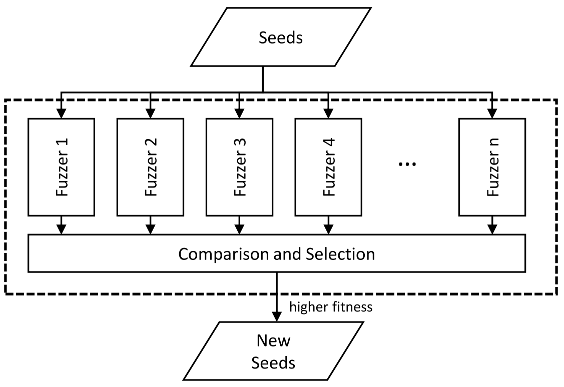


Fig. 8. Parallel fuzzing.

For instance, a user can configure the active termination condition as: *(T and i) or d*, which means that the fuzzing process will terminate when the desired delay d is achieved, or the resource limitation (running time T or cycles i) is reached. This allows users to adjust fuzzing effort based on their computing capabilities.

Passive Termination: A passive termination usually denotes a local optimum, which is an undesirable situation, where the fuzzer is stuck at a mutation "dead-end," i.e., further mutations cannot yield a better result. Consider this basic fuzzing configuration as an example.

1) Test circuit $\mathbb{P}$.
2) Fitness function $\mathcal{F}$: Naive Maximum.
3) Mutation Strategy $\mathcal{M}$: Sequential bit flip.

Under the Mutation Strategy $\mathcal{M}$, each bit of the input seed will be sequentially flipped to generate new patterns. Then, the fitness function $\mathcal{F}$ will keep the input seed with the maximum delay. However, assume that the delay of all newly generated patterns is less than or equal to that of (original) seeds, which suggests that under this mutation strategy, it is not possible to reach a higher delay value. Thus, the fuzzing process meets a passive termination condition and will terminate.

The passive termination condition is undesirable, especially when it appears before an active termination condition. Thus, we develop various mutation strategies and fitness functions to reduce the risks of being stuck at the local optimum.

In sum, with the two types of termination conditions above, the fuzzing process will be terminated when either active or passive conditions is met. This is to efficiently arrange the resource usage and avoid unnecessary computation waste.

F. Parallel Fuzzing

To maximize the fuzzing efficiency, we also develop parallel fuzzing strategies as shown in Fig. 8, which, instead of running only one mutation at each cycle, it runs multiple mutations in parallel. Each mutation works independently and generates patterns that will be added to the seed pool. Parallel fuzzing further reduces the risk of reaching passive termination condition as even if one of the mutations reaches the condition, there are others alive.

TABLE VIII
DELAY INCREASE COMPARED TO RANDOM TESTING

	INT_ADD	FP_ADD	INT_MUL	FP_MUL	INT_MAC
DFuzz-d	2.91%	2.33%	1.98%	1.71%	1.31%
DFuzz-nd	5.97%	4.08%	4.49%	3.36%	2.61%
DFuzz-p	8.30%	8.11%	5.45%	4.61%	3.88%

*G. **DFuzz** Performance*

In this section, we describe the **DFuzz** performance in generating high-quality input patterns for DTA. DTA is recently increasingly used in timing speculation research when designers intend to overclock the circuit to find the fastest clock speed at which there are no or little timing violations [11], [37], [40], [50]. Their DTA relies either on random stimuli [11], [37] or representative application workload [40], [50]. Our experiments on the Sobel and Gaussian filters show that the application workload has a strong data locality and, hence, only sensitize relatively limited paths compared to random workload. Therefore, we compare **DFuzz**-generated patterns with the random workload.

Fig. 9 illustrates the comparison between **DFuzz** and random pattern generation (RAND). The baseline RAND strategy for comparison randomly generates patterns. For **DFuzz** here, we implement two mutation strategies: 1) deterministic (**DFuzz**-d) and 2) nondeterministic (**DFuzz**-nd). The delay measurement is accelerated by **DEVoT**. Based on Fig. 9, we can observe several important facts. First, within 200 s, **DFuzz**-d has successfully generated patterns that cause longer delays ranging from 1.31%–2.91% higher than RAND. However, deterministic strategies reaches passive termination condition at around 400 s. In contrast, **DFuzz**-nd is alive for a longer period of time than the **DFuzz**-d. The delay sensitized by **DFuzz**-nd generated patterns can still increase after a relatively longer period and ultimately reaches 2.61%–5.97% higher than RAND. We can also observe that it is easy for RAND to meet the termination over a short period of time while **DFuzz** can maintain its liveness and delay increase for a longer period of time. This indicates the ability of **DFuzz** in generating high-quality patterns for DTA.

H. Parallel Fuzzing Performance

This section describes the improvement after using parallel processing (**DFuzz**-p). Fig. 10 shows the performance improvement of **DFuzz**-p compared to RAND. As stated before, **DFuzz**-p runs multiple fuzzing processes in parallel. For fair comparison, we also implement four parallel random pattern generation. As shown in Fig. 10, **DFuzz**-p solidly outperforms RAND. In particular, **DFuzz**-p is able to maintain a reasonably robust growth for the entire 1000-s testing, and can achieve 3.88%–8.30% higher delay than the random testing.

We also compared **DFuzz**-p performance with the performance of test patterns generated by the state-of-the-art ATPG delay test generation tool (marked as the dashed line in Fig. 10). The ATPG tool we used for comparison is a Boolean Satisfiability-based (SAT) method that aims at generating test patterns for transition delay faults. As discussed in [23], due to

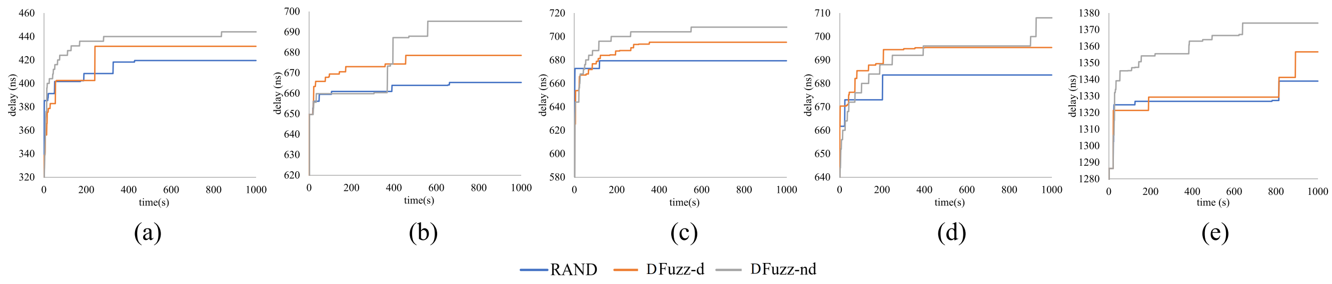


Fig. 9. Performance of **DFuzz** with nonparallel strategies. (a) INT_ADD. (b) INT_MUL. (c) FP_ADD. (d) FP_MUL. (e) INT_MAC.

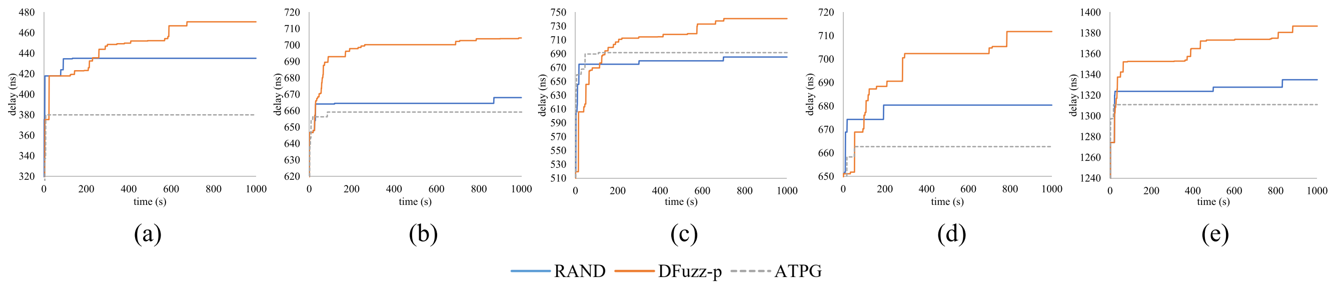


Fig. 10. Performance of **DFuzz** with parallel fuzzing strategies versus RAND & ATPG. (a) INT_ADD. (b) INT_MUL. (c) FP_ADD. (d) FP_MUL. (e) INT_MAC.

TABLE IX
DELAY INCREASE COMPARED TO ATPG

	INT_ADD	FP_ADD	INT_MUL	FP_MUL	INT_MAC
DFuzz-p	21.23%	7.13%	6.86%	7.41%	5.78%

the excessive computational complexity related to path delay fault ATPG, most of the timing-related defects in industry are modeled using transition faults, where the number of faults is linear in the number of gates. Once the test patterns are generated using the SAT solver, we then measure the delay of these test patterns using our **DEVoT** model. Experimental results show that although ATPG can generate patterns triggering higher delays at the beginning of the testing, it quickly converges and stops increasing after short period. In fact, for most FUs, ATPG is unable to find new patterns after certain runtime. In addition, except for FP_ADD, delays of patterns generated by ATPG are lower than RAND as the testing process evolves over time. **DFuzz**-p shows superior performance to ATPG in all the FUs. We summarize the performance improvement by **DFuzz**-p over ATPG in Table IX.

In summary, Table VIII shows that **DFuzz** outperforms RAND under all mutation strategies. Among them, even the simplest **DFuzz**-d can achieve 1.31%–2.91% longer delay than RAND. Furthermore, **DFuzz**-nd can achieve 2.61%–5.97% longer delay, which is higher than **DFuzz**-d. This is largely due to that with more sophisticated mutation strategies, we have better chance to avoid stuck at local optimum. Last but not least, **DFuzz**-p can even improve the delay to 3.88%–8.30% than RAND. This delay increase can significantly impact the timing speculation design. If the clock period is set using RAND with relatively low delay sensitization, there is a high chance of timing violation since there are more input patterns with longer delay not identified. On the contrary, **DFuzz** can achieve higher delay sensitization and identify more critical patterns, which can better guide the clock frequency setting in timing speculation, reducing the possibility of timing violations during runtime.

VII. Discussion

Usage: DEVoT can be used by different communities. First, in circuit testing and reliability community, **DEVoT** can assist circuit designers perform early design space exploration quickly. Second, in the approximate computing community, **DEVoT** can assist software developers assess their program resilience to hardware approximations without access/knowledge to circuit simulation.

Scope: We focus on arithmetic FUs as they often represent timing-critical parts in a pipeline [11], [48] and they are the main targets in approximate computing [38], [41], [44]. Our future work will incorporate other circuit types such as memory.

Learning Method: While the selection and tuning of learning algorithm is important to achieve good accuracy, it is not the main focus of this article. We leave this direction open to follow up research, e.g., applying more advanced learning algorithms.

VIII. Conclusion

We proposed **DEVoT**, a supervised learning model that can predict the timing delay/errors of FUs under variations in operating conditions and workload. We performed extensive dynamic delay characterization under a wide range of operating conditions and extracted useful features from the input data to predict the dynamic delay, based on which we can predict timing errors across different clock speeds. We applied RF methods to train **DEVoT**. On average, across 100 operating conditions, three clock speeds, four FUs, and three datasets, **DEVoT** achieves on average less than 2% relative deviation on delay prediction from ground truth and is 100× faster than the gate-level simulation. **DEVoT** can obtain an average prediction accuracy at 98.04% in predicting timing errors, significantly higher than baseline models. We further used **DEVoT** to estimate the application quality for two image-processing applications and **DEVoT** can estimate the quality with a 97% accuracy. Fuzzing-generated input patterns based on **DEVoT** can increase the sensitized delay by up to 8.3% compared to random patterns and 21.23% to patterns generated by ATPG. Our future work focuses on developing error models for more variation parameters such as process variations and apply them on other circuit types.

References

[1] *AMD App SDK*. [Online]. Available: https://developer.amd.com/tools-and-sdks/

[2] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," *IEEE Trans. Softw. Eng.*, vol. 45, no. 5, pp. 489–506, May 2019.

[3] S. Borade, B. Nakiboğlu, and L. Zheng, "Unequal error protection: An information-theoretic perspective," *IEEE Trans. Inf. Theory*, vol. 55, no. 12, pp. 5511–5539, Dec. 2009.

[4] K. Bowman *et al.*, "Energy-efficient and metastability-immune resilient circuits for dynamic variation tolerance," *IEEE J. Solid-State Circuits*, vol. 44, no. 1, pp. 49–63, Jan. 2009.

[5] M. Bushnell and V. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed Signal VLSI Circuits*, vol. 17. New York, NY, USA: Springer, 2004.

[6] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 33–52, 2013.

[7] K. Chae, S. Mukhopadhyay, C.-H. Lee, and J. Laskar, "A dynamic timing control technique utilizing time borrowing and clock stretching," in *Proc. IEEE Custom Integr. Circuits Conf. (CICC)*, 2010, pp. 1–4.

[8] K.-T. Cheng and A. Krstic, "Current directions in automatic test-pattern generation," *Computer*, vol. 32, no. 11, pp. 58–64, 1999.

[9] H. Cherupalli and J. Sartori, "Graph-based dynamic analysis: Efficient characterization of dynamic timing and activity distributions," in *Proc. ICCAD*, 2015, pp. 41–54.

[10] H. Cho, L. Leem, and S. Mitra, "ERSA: Error resilient system architecture for probabilistic applications," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 31, no. 4, pp. 546–558, Apr. 2012.

[11] J. Constantin, L. Wang, G. Karakonstantis, A. Chattopadhyay, and A. Burg, "Exploiting dynamic timing margins in microprocessors for frequency-over-scaling with instruction-based clock adjustment," in *Proc. DATE*, 2015, pp. 381–386.

[12] T. M. Cover and P. E. Hart, "Nearest neighbor pattern classification," *IEEE Trans. Inf. Theory*, vol. IT-13, no. 1, pp. 21–27, Jan. 1967.

[13] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with flopoco," *IEEE Design Test Comput.*, vol. 28, no. 4, pp. 18–27, Jul./Aug. 2011.

[14] D. Ernst *et al.*, "Razor: A low-power pipeline based on circuit-level timing speculation," in *Proc. MICRO*, 2003, pp. 7–18.

[15] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *Proc. ASPLOS*, 2012, pp. 301–312.

[16] F.-F. Li, R. Fergus, and P. Perona, "Learning generative visual models from few training examples: An incremental Bayesian approach tested on 101 object categories," *Comput. Vis. Image Understand.*, vol. 106, no. 1, pp. 59–70, 2007.

[17] M. Fojtik, D. Fick, Y. Kim, N. Pinckney, D. Harris, D. Blaauw, and D. Sylvester, "Bubble razor: An architecture-independent approach to timing-error detection and correction," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2012, pp. 488–490.

[18] K. Fuchs, F. Fink, and M. H. Schulz, "DYNAMITE: An efficient automatic test pattern generation system for path delay faults," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 10, no. 10, pp. 1323–1335, Oct. 1991.

[19] P. Godefroid, "Random testing for security: Blackbox vs. whitebox fuzzing," in *Proc. ACM 2nd Int. Workshop Random Test.*, 2007 p. 1.

[20] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proc NDSS*, vol. 8, 2008, pp. 151–166.

[21] P. Gupta *et al.*, "Underdesigned and opportunistic computing in presence of hardware variability," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 32, no. 1, pp. 8–23, Jan. 2013.

[22] R. K. Gupta, S. Mitra, and P. Gupta, "Variability expeditions: A retrospective," *IEEE Des. Test*, vol. 36, no. 1, pp. 65–67, Feb. 2019.

[23] S.-Y. Ho *et al.*, "Automatic test pattern generation for delay defects using timed characteristic functions," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2013, pp. 91–98.

[24] X. Jiao, M. Luo, J.-H. Lin, and R. K. Gupta, "An assessment of vulnerability of hardware neural networks to dynamic voltage and temperature variations," in *Proc. ICCAD*, 2017, pp. 945–950.

[25] X. Jiao, Y. Jiang, A. Rahimi, and R. K. Gupta, "SLoT: A supervised learning model to predict dynamic timing errors of functional units," in *Proc. Design Autom. Test Europe*, 2017, pp. 1183–1188.

[26] X. Jiao, D. Ma, W. Chang, and Y. Jiang, "LEVAX: An input-aware learning-based error model of voltage-scaled functional units," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 12, pp. 5032–5041, Dec. 2020.

[27] V. B. Kleeberger, P. R. Maier, and U. Schlichtmann, "Workload-and instruction-aware timing analysis-the missing link between technology and system-level resilience," in *Proc. 51st ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, 2014, pp. 1–6.

[28] E. Krimer, P. Chiang, and M. Erez, "Lane decoupling for improving the timing-error resiliency of wide-SIMD architectures," in *Proc. ISCA*, 2012, pp. 237–248.

[29] A. Krstic, Y.-M. Jiang, and K.-T. Cheng, "Pattern generation for delay testing and dynamic timing analysis considering power-supply noise effects," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 3, pp. 416–425, Mar. 2001.

[30] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2018, pp. 1–8.

[31] C. J. Lin and S. M. Reddy, "On delay fault testing in logic circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst*, vol. CAD-6, no. 5, pp. 694–703, Sep. 1987.

[32] N. Linial *et al.*, "Constant depth circuits, Fourier transform, and learnability," *J. ACM*, vol. 40, no. 3, pp. 607–620, 1993.

[33] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels," in *Proc. ACM Int. Conf. Ob. Oriented. Program. Syst. Lang. Appl.*, 2014, pp. 309–328.

[34] P. Ndai, N. Rafique, M. Thottethodi, S. Ghosh, S. Bhunia, and K. Roy, "Trifecta: A nonspeculative scheme to exploit common, data-dependent subcritical paths," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 18, no. 1, pp. 53–65, Jan. 2010.

[35] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Oct. 2011.

[36] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2017, pp. 2155–2168.

[37] A. Rahimi, L. Benini, and R. K. Gupta, "Analysis of instruction-level vulnerability to dynamic voltage and temperature variations," in *Proc. DATE*, 2012, pp. 1102–1105.

[38] A. Rahimi, L. Benini, and R. K. Gupta, "Hierarchically focused guardbanding: An adaptive approach to mitigate PVT variations and aging," in *Proc. IEEE DATE*, 2013, pp. 1695–1700.

[39] A. Rahimi, L. Benini, and R. K. Gupta, "Application-adaptive guardbanding to mitigate static and dynamic variability," *IEEE Trans. Comput.*, vol. 63, no. 9, pp. 2160–2173, Sep. 2014.

[40] S. Roy and K. Chakraborty, "Predicting timing violations through instruction-level path sensitization analysis," in *Proc. ACM 49th Annu. Design Autom. Conf.*, 2012, pp. 1074–1081.

[41] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," in *Proc. PLDI*, 2011, pp. 164–174.

[42] S. R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas, "VARIUS: A model of process variation and resulting timing errors for microarchitects," *IEEE Trans. Semicond. Manuf.*, vol. 21, no. 1, pp. 3–13, Feb. 2008.

[43] J. Sartori, J. Sloan, and R, Kumar, "Stochastic computing: Embracing errors in architectureand design of processors and applications," in *Proc. CASES*, 2011, pp. 135–144.

[44] G. Tziantzioulis, A. M. Gok, S. M. Faisal, N. Hardavellas, S. O. Memik, and S. Parthasarathy, “b-hive: A bit-level history-based error model with value correlation for voltage-scaled integer and floating point units,” in *Proc. DAC*, 2015, pp. 1–6.

[45] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. R. Kaeli, “ Multi2Sim: A simulation framework for CPU-GPU computing ,” in *Proc. PACT*, 2012, pp. 335–344.

[46] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan, “MACACO: Modeling and analysis of circuits for approximate computing,” in *Proc. ICCAD*, 2011, pp. 667–673.

[47] S. H. Wu, A. Tetelbaum, and L.-C. Wang, “How does inverse temperature dependence affect timing sign-off,” in *Emerging Technologies and Circuits*. Dordrecht, The Netherlands: Springer, 2010, pp. 179–189.

[48] J. Xin and R. Joseph, “Identifying and predicting timing-critical instructions to boost timing speculation,” in *Proc. MICRO*, 2011, pp. 128–139.

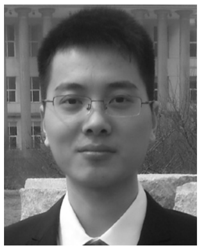
[49] M. Zalewski. (2017). *American Fuzzy Lop*. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[50] J. J. Zhang and S. Garg, “Bandits: Dynamic timing speculation using multi-armed bandit based optimization,” in *Proc. Conf. Design Autom. Test Europe*, 2017, pp. 922–925.

[51] X. Jiao, D. Ma, W. Chang, and Y. Jiang, “TEVoT: Timing error modeling of functional units under dynamic voltage and temperature variations,” in *57th ACM/IEEE Design Autom. Conf. (DAC)*, 2020, pp. 1–6.

Dongning Ma (Graduate Student Member, IEEE) received the Bachelor of Engineering degree in automation from the School of Advanced Engineering, University of Science and Technology Beijing, Beijing, China, in 2018. He is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, Villanova University, Villanova, PA, USA.

His research interests include approximate computing, embedded systems, machine learning, and novel computing schemes.



Xinqiao Zhang (Graduate Student Member, IEEE) received the B.S. degree in automation from Northeastern University, Shenyang, China, and the M.S. degree in electrical engineering from San Diego State University, San Diego, CA, USA, in 2017. He is currently pursuing the Ph.D. degree in electrical and computer engineering jointly with the University of California at San Diego, San Diego, and San Diego State University.

His current research interests include hardware trojan detection, adversarial machine learning, and VLSI physical design.

Ke Huang (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in electrical engineering from Université Grenoble Alpes, Grenoble, France, in 2006, 2008, and 2011 respectively.

He was a Postdoctoral Research Associate with the University of Texas at Dallas, Richardson, TX, USA, from 2012 to 2014. In 2014, he joined the Department of Electrical and Computer Engineering, San Diego State University, San Diego, CA, USA, where he is currently an Associate Professor. He has published over 30 journal and conference papers. His research focuses on very large-scale integration testing and security, computer-aided design of integrated circuits, and intelligent vehicles.

Dr. Huang was a recipient of the Best Paper Award from the 2015 IEEE VLSI Test Symposium and the Best Paper Award from the 2013 Design Automation and Test in Europe conference. He has served as a Program Committee Member for various IEEE conferences, and as a Guest Editor for the *Journal of Electronic Testing Theory and Applications* (Springer).

Yu Jiang received the B.S. degree in software engineering from the Beijing University of Post and Telecommunication, Beijing, China, in 2010, and the Ph.D. degree in computer science from Tsinghua University, Beijing, in 2015.

He is an Assistant Professor with the School of Software, Tsinghua University. He was a Postdoctoral Researcher with the University of Illinois at Urbana–Champaign, Champaign, IL, USA. His current research interests include domain-specific modeling, formal verification, and their applications in embedded systems, safety analysis, and assurance of CPS.

Wanli Chang (Member, IEEE) received the bachelor’s degree (First-Class Hons.) from Nanyang Technological University, Singapore, in 2008, and the Ph.D. degree in electrical and computer engineering from the Technical University of Munich, Munich, Germany, in 2017.

He is with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, China, and also with the Department of Computer Science, University of York, York, U.K. His research interest is on trusted and resource-aware cyber–physical systems.

Dr. Chang won the Departmental Best Dissertation Award from the Technical University of Munich. He is serving on the TPC of premium conferences on embedded and real-time systems, as well as design automation, including DAC, ICCAD, DATE, RTSS, EMSOFT, CODES+ISSS, and LCTES.

Xun Jiao received the dual bachelor’s degree from the joint program of the Beijing University of Posts and Telecommunications, Beijing, China, and Queen Mary University of London, London, U.K., in 2013, and the Ph.D. degree from the Department of Computer Science and Engineering, University of California at San Diego, San Diego, CA, USA, in 2018.

He is an Assistant Professor with the Department of Electrical and Computer Engineering, Villanova University, Villanova, PA, USA. His research interests include embedded systems, design automation, and brain-inspired computing.

Dr. Jiao is an Associate Editor of IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS and a TPC Member of DAC, GLSVLSI, and LCTES.