WILD: A Workload-Based Learning Model to Predict Dynamic Delay of Functional Units

Xun Jiao[‡], Yu Jiang[§], Abbas Rahimi[*], and Rajesh K. Gupta[‡]
[‡]Department of Computer Science and Engineering, UC San Diego, La Jolla, CA, USA
[§] School of Software, Tsinghua University, Beijing, China
[*]Department of Electrical Engineering and Computer Sciences, UC Berkeley, Berkeley, CA, USA
{xujiao, gupta}@cs.ucsd.edu, jiangyu198964@gmail.com, abbas@eecs.berkeley.edu

***Abstract*—Dynamic critical path analysis in modern processors is needed to reduce margins typically determined by the static timing analysis. Dynamic path analysis, however, is cost-prohibitive. In this paper, we propose `WILD`, a supervised learning model to predict dynamic delay of functional units (FUs) based on the input workload during execution. We measure the dynamic delay using switching activity generated through gate-level simulation of a post place-and-route design in TSMC 45nm process. We then look for 'features' in the input data that influence dynamic path sensitization. Using these features we apply a logistic regression (LR) method to construct a predictive model trained and tested using three datasets: random, Sobel filter and Gaussian filter. We classify dynamic delay into five distinct classes. For a given test input, `WILD` predicts the class of output dynamic delay. On average across several FUs, 98.0% of `WILD` predictions are consistent with gate-level simulation. Using `WILD`-directed dynamic frequency scaling can improve instruction-level performance by 13%–44% compared to the state-of-the-art instruction-level timing model.**

I. Introduction

To ensure error-free operation, traditional circuits are designed with a conservative timing margin based on critical path computed from a multi-corner worst-case analysis at design time derived through static timing analysis. Although a synchronous circuit design is safe with static timing constraints using worst case critical path delay, in reality, the critical path is rarely sensitized during execution thus resulting in unnecessary loss of performance. This problem is further exacerbated by increased variability in advanced processes, caused by process, voltage, temperature and aging (PVTA) effects.

To improve the system performance, better-than-worst-case (BTWC) design methods have been explored (e.g., [4]). These methods aim to reduce the needs for large timing margins without compromising safe operation by improved static characterization based on part-specific information and/or runtime sensing [2]. A typical approach is to use frequency overscaling to reduce timing margin, and use recovery techniques to correct the timing violations induced by such frequency overscaling [4], [24]. Such methods are effective, but can incur silicon overhead for online monitoring. Further, these methods impose recovery penalty in the case of timing violations.

An alternative and less intrusive technique is to predict the timing delay in advance and then prevent timing violations by adjusting the operating frequency accordingly. Several instruction-level timing prediction models have been proposed to measure the timing delay of instructions using gate-level simulations [22]. A timing error rate prediction model for functional units (FUs) has been proposed to reduce guardband hierarchically by obtaining hardware PVTA variation information [20]. However, these instruction-level models assume a worst case scenario, that overlooks the effect of input operands on the path sensitization behavior, leading to a less efficient or pessimistic modeling of hardware timing.

There is, of course, a correlation between the input workload and timing violations because of its direct effect on dynamic path sensitization. During execution, the sensitized paths strongly vary with different input workload [18]. This is seen in different instruction-level timing delay as a function of the operands to instructions. This paper explores use of a workload-dependent predictive model for instruction-level timing managements, where we face the following challenges:

Challenge 1: we need to measure the dynamic timing delay of FUs at each cycle under different input operands. Based on the measurement results and input workload, we need to investigate the specific source factors that could affect the dynamic circuit delay. This requires a trial-and-error iterative process of varying input parameters and examining its corresponding output.

Challenge 2: we have no prior knowledge of the circuit structure and it is even unclear what factors could affect dynamic path sensitization. In general, under cryptographic assumptions Probably Approximately Correct (PAC) learning of Boolean circuits is hard [16] even under uniform distribution over the inputs [17].

Challenge 3: for model generation purpose, we need to extract the useful features from a large set of input operands, to train the circuit delay model over a large input space of m input bits. Note that each bit position might have different weights on dynamic timing delay because the bit significance varies.

Proposed approach: To overcome these challenges, we propose `WILD`, a supervised learning model, to predict the dynamic delay of FUs based on input workload or operands. First, we extracted useful input features from input workload by analyzing the variations of circuit timing delay, measured

978-1-5090-5142-7/16/$31.00 ©2016 IEEE

dynamically each cycle under different input operands. We employ a switching activity file obtained from a post-layout gate-level simulation on TSMC 45nm technology. Then, we use supervised learning methods to construct and train the model, evaluat and compare with the baseline modeling in terms of prediction accuracy. We evaluated various commonly used machine learning techniques and finally chose logistic regression as our modeling method due to its high prediction accuracy and efficient computing time. Furthermore, we have applied our prediction model to three different datasets, random data, Sobel filter and Gaussian filter, and achieve prediction accuracy ranging between 96.2–99.8%. Using **WILD**-directed dynamic frequency scaling (DFS), the average instruction-level timing delay could be reduced for three different instructions (Int_ADD, Int_Mul and FP_MUL). In addition, our model execute 60X faster compared with gate-level simulation to compute dynamic delay for 200k data.

Contributions: This paper makes the following contributions:

- We develop an accurate dynamic timing delay measurement methodology based on switching activity derived through gate-level simulation to analyze the sources of delay variations and the effect of input workload on circuit-level timing delay.
- We propose a methodology to construct automatic models for dynamic delay timing prediction using supervised learning methods. To our best knowledge, this is the first prediction model of functional units on timing delay that can capture the dynamic path sensitization behavior under different input operands.
- The evaluation results demonstrate the robustness and accuracy of **WILD** in prediction and its effectiveness in system performance increase. We profile the test input workload from real world applications and achieve 96.2–99.8% prediction accuracy, which is 3.6X and 1.5X higher compared to two baseline models. Further, by using **WILD**-directed DFS, the instructions can achieve 13–44% operation speed up compared with the state-of-art instruction-level timing model directed DFS [5] [22].

II. Background

We evaluate four learning methods here that have been chosen for their increased sophistication and practical use: k-nearest neighbor (k-NN), support vector machine (SVM), logistic regression (LR) and decision tree (DT) classifiers [3]. k-NN performs classification based on the distance between feature vectors, treating every bit position equally, regardless of their bit significance difference. The opposite goes to SVM and LR, which assign the weight to each bit location and predict classification based on the weighted numerical results. DT learns a decision rule to determine prediction process, which has a superior interpretability. We expect k-NN will have low prediction accuracy because of its ignorance of bit location effect. DT could easily suffer from overfitting. SVM is expected to have high training time compared to the others. The four methods would be deeply evaluated in experiment section and some concepts are introduced as below.

k-NN is a non-parametric method for classification and regression. It predict the timing delay class membership given an input vector x if majority of the k nearest neighbors of x in the dataset $\mathcal{D}$ map to that class. We select k=5 in this work to balance the approximation error and estimation error. Thus, it is naturally extensible to multi-classification. Although providing useful theoretical properties, k-NN, as an instance-based learning, or lazy learning method, often suffers the sub-par generalization performance (i.e., performance on new data) when available training labeled data is limited. Besides, it also has the problems of appropriate feature normalization and scaling. More importantly, we expect k-NN would have a bad performance on the timing delay prediction since the timing delay is affected differently by different bit positions while k-NN assigns the equal weight to different bit positions. To address these problems, we consider two methods that learn weights w on the bit-level features that maximize the classification accuracy.

SVM is inherently a binary classifier. It could implement multi-classification problems in different ways. Here we use the most common one-against-rest approach because of its computational efficiency and high interpretability. Given C as the number of classes, it constructs C independent classifiers and calculates the probability of each class against the rest and chooses the class with highest probability. For an unseen input x, we compute the probability of each class and choose the one with highest probability. Given labels y_i for the N training data points x_i, SVMs learn w based on the following large margin optimization problem:

$$\begin{aligned} \min_{w,\eta,\rho} \quad & \frac{1}{2}||w||^2 + \frac{1}{N}\sum_i \eta_i - \nu\rho \\ \text{s.t.} \quad & y_i(w \cdot x_i - w_0) \geq \rho - \eta_i \end{aligned} \tag{1}$$

The weights are learned to maximize the margin (η_i) by which the training data are correctly classified. We use the popular Radial Basis Function (RBF) kernel to map the input training data to a higher dimensional kernel space to enable non-linear classification.

LR also can implement the multi-classification problem using a one-against-rest approach, where it fits one classifier per class and calculates the probability of each class against the rest. In the logistic regression classifier, we learn weights that can maximize the probability of labeled class on the training data $\mathcal{D}$. For an unseen input x, the probability of each class is computed and the one with highest probability is chosen, where the probability function is given by

$$F(x) = \frac{1}{1 + e^{-w \cdot x}} \tag{2}$$

DT is also a non-parametric supervised learning method and could be naturally extended to multi-classification. It predicts the output class given an input x by learning the decision rules derived from training examples. However, decision tree can easily suffer from the overfitting problem when it becomes very deep as it learns a lot of irregular pattern with a large

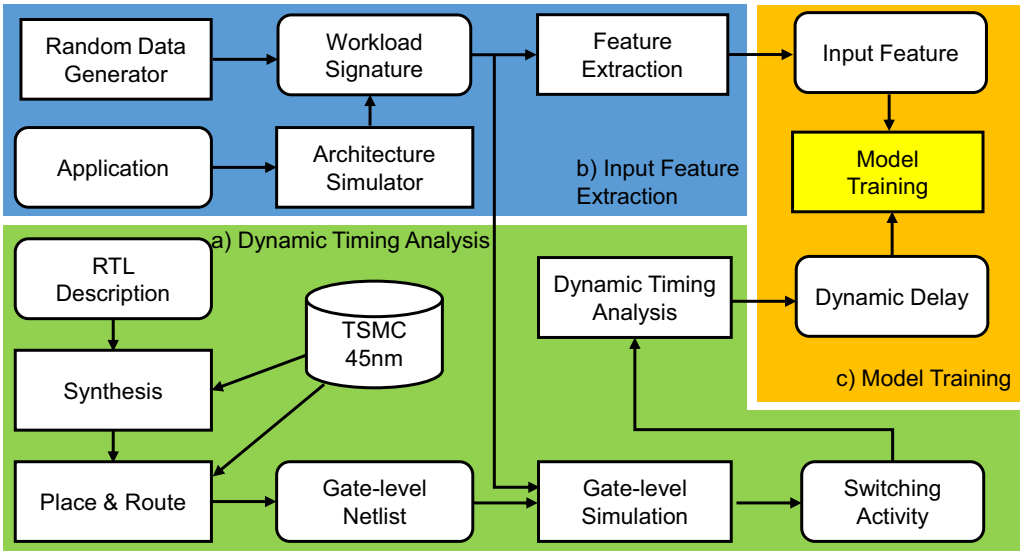

Fig. 1. **WILD** model overview with three key phases: a) Dynamic Timing Analysis to measure the dynamic delay under different input workload;b) Input Feature Extraction to extract useful 'features' from random data and real-world application input data; c) Model Training to use supervised learning model with extracted useful 'features' and dynamic delay as labeled output to train the model.

variance. Therefore, we limit the number of instances for each leaf node to prevent overfitting.

III. Problem Formulation and WILD Model

Problem formulation: We represent the dynamic timing delay of a FU as a function of its input workload. More specifically, we abstract a circuit as a mapping from an input space $\mathcal{I}$ consisting of m input bits, to create an input I. Defining $\psi(I)$ as the timing delay of the circuits, our goal is to learn (an approximation) of ψ given uniform samples from the 2^m possible input bits. However, we have no prior knowledge of the structure of the delay function ψ. Thus, we classify the input operands to map to different output delay by using a classification method. We define the output timing delay into different classes, and map the input workload to one of the classes, forming a multi-classification problem.

WILD Model: It is comprised of three phases as shown in Fig. 1: *Dynamic Timing Analysis, Input Feature Extraction* and *Model Training*. a) The *Dynamic Timing Analysis* phase implements the standard ASIC flow and uses gate-level simulation to generate switching activity file. Then, our Python-written dynamic timing analysis script will analyze the switching activity file to generate the dynamic delay under different input workload. b) In the *Input Feature Extraction*, we generate the input training data in two ways: using a random data generator and profiling of FU input operands in real-world applications using architectural-level simulator. Then, the workload signature of training data is pre-processesed and useful features are extracted from the training data, such as bit locations and input history, which are then incorporated into model training. c) In the *Model Training* phase, the model is trained with the previous collected data using different supervised learning algorithms. We classify the output dynamic delay into different classes and the model will predict the class to which the output delay belongs for a given input data. More details about the three phases are illustrated as follows.

A. Dynamic Timing Analysis

We focus on three different types of FUs, 32-bit integer adder and multiplier, and 32-bit single-precision floating point multiplier. The floating point units (FPUs) are compatible with IEEE-754 standard, and can provide more complex circuit structures compared to their integer counterparts. We vary the circuit structures not only by function types but also by data types to assess the robustness of our model.

We use FloPoCo [8] to generate the synthesizable VHDL codes of FUs with wrapper at input and output ports. *Synopsys Design Compiler* is used to synthesize the VHDL codes and *Synopsys IC Compiler* is used to do place&route to generate post-layout netlist in TSMC 45nm technology. *Synopsys PrimeTime* is used to do static timing analysis to generate Standard Delay Format (SDF) file. Then, we use *Mentor Graphics Modelsim* to do SDF-back-annotation gate-level simulation to generate value change dump (VCD) file as a switching activity file. The stimuli input comes from two sources: random data generator script written in Python and the application input data profiled using *Multi2Sim* [26], a cycle-accurate CPU-GPU heterogeneous architectural simulator.

Next, unlike static timing analysis which can only give us the static timing of path delay, we use the switching activity file to do the dynamic timing analysis. The VCD file records the toggled nets at each cycle thus giving us the dynamic path sensitization information. To extract the dynamic delay based on sensitized critical path, we are only interested in the endpoints of every timing path. We run the simulation at a relatively slow clock period to make sure there is no

TABLE I
FIVE CLASSES OF DYNAMIC DELAY (PS).

$500 > \text{delay} \geq 0$	$1000 > \text{delay} \geq 500$	$1500 > \text{delay} \geq 1000$	$2000 > \text{delay} \geq 1500$	$2500 > \text{delay} \geq 2000$
C_{ex_low}	C_{low}	C_{med}	C_{high}	C_{ex_high}

timing violation. For each clock cycle, we use the last toggle event time of the input pin of all sequential elements (flip flop, registers, etc) to subtract the last positive clock edge arrival time to get the maximum delay at that cycle. For example, at cycle N the positive clock edge occurs at time t, and the very last toggled event at the data input pin of all sequential elements occurs at time t', then the dynamic delay at this cycle is $t' - t$. We run a large simulation and probe all toggled events at data input pins of sequential elements, and then parse the VCD file using our dynamic timing analysis tool that can provide us the dynamic delay at each cycle under different input workload. Note that when the input operands are the same for two consecutive cycles, there is no toggled nets, resulting a zero dynamic delay.

B. Input Feature Extraction

Having said before, there are extremely high number of possible input combinations. Given two 32-bit operands, there are 2^{64} different combinations. Thus, it is not feasible to apply all 2^{64} input patterns for training. To cover a large range of input space, we use the homogeneous distribution of two operands over 2D input space used in [25]. By applying these training input to the *dynamic timing analysis* module, we obtain the dynamic delay corresponding to each input workload. Then, for the training purpose, we need to find out the useful input features, i.e., the source factors which determine the dynamic delay. Intuitively, the current input workload directly affects the path sensitization. However, the preceding history input might also affect path sensitization of current cycle because the preceding input will set a state of the circuit and affect the signal transition between two cycles. In order to investigate the effect of history input workload, we use a trial-and-error process to iteratively vary the history input workload while keeping the current input fixed. We set the experiments as follows:

- Scenario 1: we only fix the current input while varying the immediate preceding input. We use this to evaluate the effect of immediately preceding input.
- Scenario 2: we fix both current and immediately preceding input while varying the preceding input of the immediately preceding one. We use this to evaluate effect of deeper history.

We perform 100K gate-level simulation to assess the effect of workload history and it turns out the dynamic delay of scenario 1 varies without irregularly while the scenario 2 results in constant dynamic delay. Therefore, we conclude that only the immediately preceding input will have effect on the dynamic delay of current cycle. This is expected as only the immediately preceding input and current input determine the signal transition which determines the path sensitization.

Next, we need to do data preprocessing to clean the training data. First of all, the decimal format of input data needs to be converted into binary vector representation. The reason behind this conversion is that, the circuit uses 32-bit vectors as input format and the 0/1 value at each bit location could affect different paths thus inferring different path sensitization behaviors. Meanwhile, the decimal format cannot precisely reflect the significance of each bit position. Therefore, we convert the decimal format to binary format, e.g., 0.5 is converted to 00111111000000000000000000000000. Actually, LR and SVM emphasizes this bit significance naturally, where they assign different weights to different bit locations to optimize the object function. Meanwhile, k-NN cannot distinguish the difference among bit locations because it treats each bit location equally weighted to determine the distance between feature vectors. The next step is to clean the training data by removing the repetitive data patterns resulting the same delay, and excluding the cycles with a zero dynamic delay which could happen when both preceding cycle and current cycle have same input operands and no nets are toggled. These two scenarios need to be excluded to save meaningless training efforts. In summary, after preprocessing the training data, we need to extract the useful features of input vectors: history input and bit location, to train the model accurately and efficiently.

C. Model Training

First, the output dynamic delay is classified into five different classes. Since 2.5ns is the clean clock period under which no timing violations occur for all of our designs, we use 500ps as step size, resulting in five different classes: C_{ex_low}, C_{low}, C_{med}, C_{high} and C_{ex_high} based on their delay range as shown in Table. I. The reason of using five classes is that clock controller circuit (CGU) in DFS can only use a limited number of *phase locked loops* (PLLs) [23], each with a pre-configured fixed frequency. Empirically, in this work, we assume five PLLs are used in CGU to balance the tradeoff between frequency resolution and hardware overhead.

Then, we set the previously extracted input features, $\{x[t], x[t-1]\}$ as input feature and $C[t]$ as output labeled class, where x[t] and x[t-1] are input binary vectors at cycle t and $t-1$, and $C[t] \in \{C_{ex\_low}, C_{low}, C_{med}, C_{high}, C_{ex\_high}\}$. The input training data comes from two sources: one is from random generated data and the other is from the input operands profiled from real world applications. The input workload is then applied to gate-level simulation and dynamic timing analysis to obtain the dynamic delay value for each input workload, which is then classified into $C[t]$. With the input features and output labels, the four supervised learning methods are applied in Sec. II to construct a multi-classification model.

TABLE II
PREDICTION ACCURACY, AND TOTAL TRAINING AND TESTING TIME OF FOUR LEARNING METHODS.

method	Accuracy	Time (s)
KNN	0.932	233.35
SVM	0.975	2478.38
LR	0.974	1.82
DT	0.952	4.08

Finally, we evaluate the aforementioned four different supervised learning methods: k-NN, LR, SVM and DT by using 50K random training data and 10k random testing data across three FUs. As shown in Table II, we observe that LR is the fastest method with high prediction accuracy. DT is also fast but achieves low prediction accuracy. k-NN takes several minutes to finish with this small scale of training and testing data. Actually, when the training data size becomes 100k, k-NN takes several hours to perform classification, due to that for every given test data, k-NN needs to calculate the distance with respect to each training vector and find the nearest neighbors from the entire training space. This implies that the training size affects the classification time of k-NN. Meanwhile in LR, the size of training data does not affect classification time because LR model outputs only the weight vectors, which are then used to operate with test features. That is, the training size only affects the value of weight vectors hence the classification result, but not the classification time. Although SVM achieves highest prediction accuracy, its training and testing takes more than half an hour, which is highest among four methods. Therefore, we finally choose LR due to its high prediction accuracy and better computing efficiency. The machine learning modules are provided by Scikit-learn package written in Python [19].

D. Model Evaluation

For a given input workload, our model will predict the class to which it belongs among the five classes. We use prediction accuracy as our evaluation metric and compare this with two baseline models.

1) Evaluation Metric: We evaluate the prediction accuracy of prediction model by comparing prediction result with golden output generated by gate-level simulation:

$$prediction\_accuracy = \frac{\#matched\_cycles}{\#total\_cycles} \quad (3)$$

where $\#total\_cycles$ is the number of total simulation cycles, and $\#matched\_cycles$ is the number of cycles at which predicted result equals to golden result.

2) Comparison Methods: Since there are no previous works on predicting dynamic delay of FUs, we compare **`WILD`** against following baseline methods which can help us evaluate the true performance of our model:

- **rand:** predict the dynamic delay class among the non-empty classes which contain at least one instance randomly. Some classes might have no instance , for example, C_{ex_low} in Fig. 4, thus we ignore those empty classes.
- **naive:** use a naive class to always predict the class which contains most instances. If the dataset is heavily biased, e.g., 99% of the data belongs to one class, then even a trivial class can achieve 99% prediction accuracy by always predicting that class.

IV. EXPERIMENTAL RESULTS

In this section, we present the dynamic delay distribution of three FUs under three different input datasets. Then, we present the prediction accuracy of LR-directed model and compare with the baseline model using particular evaluation metric. Finally, we utilize LR model-directed dynamic frequency scaling (DFS) to adjust instruction-level operating frequency to achieve instruction execution speedup.

A. Experimental Setup

We choose two image processing applications from AMD APP SDK v2.5 [1], Sobel filter and Gaussian filter. The OpenCL codes of these applications are simulated by our modified version of *Multi2Sim* to profile input workloads of interested FUs. We choose 10 images in Caltech-UCSD Birds 200 vision dataset [27] as input image for these applications to profile test data. We select the operating voltage to be 0.85V and temperature to be 50°C.

B. Delay Distribution of Functional Units

We use the dynamic timing analysis described in Section III-A to investigate the dynamic timing delay of the three FUs under the datasets generated from: random data described in Section III-B, Sobel filter and Gaussian filter described in Section IV-A. Fig. 2 – Fig. 4 present the delay distribution of FUs under three different input datasets, from which we observe several important facts.

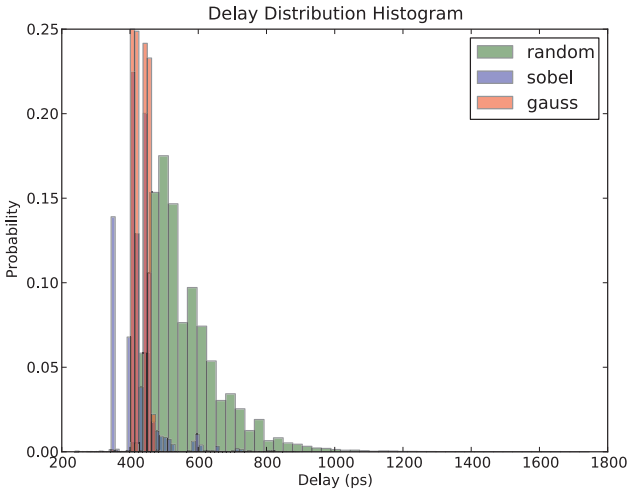

Fig. 2. Delay distribution of INT_ADD under three different input workload sets.

First, for all figures, it is clearly seen that FUs exhibit noticeably different dynamic delay under different input workload. In particular, we observe up to 5X difference of delay in INT_ADD and INT_MUL. Hence, all prior works on

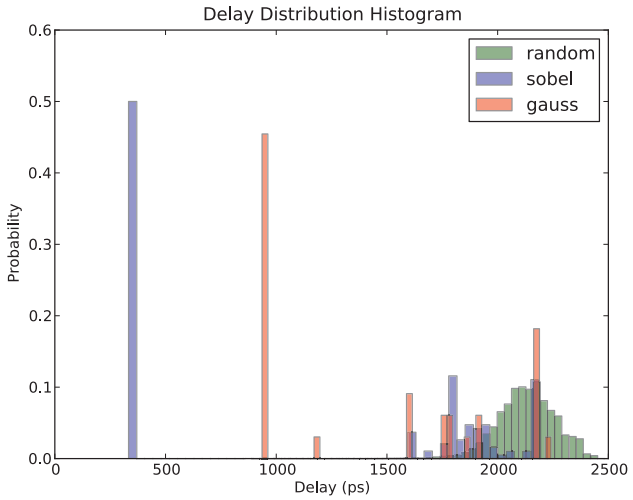

Fig. 3. Delay distribution of INT_MUL under three different input workload sets.

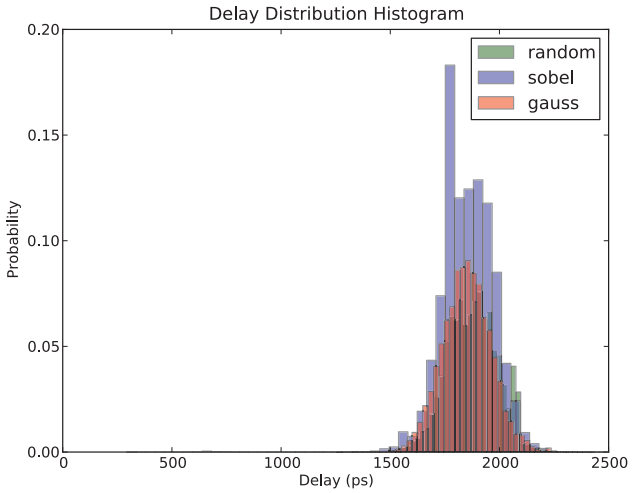

Fig. 4. Delay distribution of FP_MUL under three different input workload sets.

FUs and instruction-level timing modeling that ignore the effect of input workloads suffer from inaccuracy. Second, we observe that some FU experiences a large deviation of dynamic delay. In particular, the INT_MUL presents a non-regular delay distribution for Sobel filter and Gaussian filter while the others all present Guassian-like distribution. This is because the critical paths in INT_MUL sensitized by these two applications exhibit vastly different timing delay. Third, by observing delay behavior resulted from each dataset individually, we find majority of them exhibits a Gaussian-like distribution. This is because some critical paths are more frequently sensitized by input workload, for that the conventional design strategies tend to produce a so-called wall of slack [14], which contains a large number of near-critical paths. In particular, the INT_ADD, for example, the sum of two highest bins are accounted nearly 50% of the delay data. Fourth, comparing with random data, we can see the delay from Sobel filter and Gaussian filter exhibit a more dense distribution. This is because there is a data locality phenomena in these applications, resulting in a high commonality in path sensitization [22]. In addition, the mean value of delay of Sobel filter and Gaussian filter are less than the one derived from random data. This is because a large number of real-world input operands for these FUs are small size operands, resulting small delays. Thus, the random dataset produced the largest delay variance and hence can be used as the most representative dataset to evaluate prediction accuracy.

C. Model Prediction Accuracy

Table. III presents the prediction accuracy of dynamic delay of three FUs under three different datasets using three models: **`WILD`**, **rand** and **naive**. **`WILD`** exhibits the prediction accuracy ranging between 96.2–99.8% and achieves average prediction accuracy 98.0% over all FUs under all datasets. The **rand** model achieves average prediction accuracy at 27.2% while **naive** can achieve 63.6% on average. Thus, compared to these baseline models, **`WILD`** exhibits 3.6X and 1.5X higher prediction accuracy. We notice that, the prediction accuracy is affected by the delay distribution of different datasets. The prediction accuracy of Gaussian filter dataset is higher than that of random dataset. This is because the data in Gaussian filter dataset is so biased that its delay distribution is among a very small range then even a **naive** classifier can have a high prediction accuracy.

D. Instruction-level Timing Margin Reduction

The existing instruction-level timing models [22] [5] measure the instruction delay under the worst case assumption of input operands. However, the instruction-level timing delay during runtime depends on the actual input workload and it can be changed from time to time. Thus, the dynamic frequency scaling (DFS) enabled by the existing instruction-level models leads to pessimistic operating timing margin. DFS is an architectural technique that adjusts operating frequency on-the-fly to improve performance. To address such problem, we use **`WILD`**-directed DFS to enable a finer-grained frequency adjustment as it can provide an specific delay value for a given input workload. As a result, the circuit can run at a higher frequency compared to existing model-directed DFS.

For a given instruction, the existing models uses the worst case instruction-level timing delay measured to set the operating frequency which will be used in DFS, e.g., 826ps, 2187ps and 2438ps for Int_ADD, Int_MUL and FP_MUL in Sobel filter as shown in Fig. 2–Fig. 4. On the other hand, **`WILD`** incorporates the input workload with the instruction and predicts the class of the resulted dynamic delay and uses the upper bound of that class to set the operating frequency to provide a safe operating frequency. For example, if the predicted class is C_{ex_low}, then the DFS will use 500ps as clock period to execute the instruction. However, the predicted class could be wrong, which could fall into one of the two categories:

TABLE III
PREDICTION ACCURACY OF THREE DIFFERENT CLASSIFIER MODELS AND THREE DIFFERENT DATASETS.

	random dataset			Sobel filter dataset			Gaussian filter dataset		
	WILD	rand	naive	**WILD**	rand	naive	**WILD**	rand	naive
Int_Add	0.974	0.333	0.319	0.966	0.330	0.954	0.998	0.334	0.999
Int_Mul	0.962	0.201	0.146	0.976	0.163	0.363	0.992	0.091	0.303
FP_Mul	0.983	0.332	0.841	0.985	0.332	0.890	0.993	0.333	0.914
Average	0.973	0.288	0.435	0.975	0.275	0.736	0.994	0.253	0.739

TABLE IV
AVERAGE INSTRUCTION-LEVEL TIMING DELAY(PS) USING **WILD** COMPARED TO EXISTING INSTRUCTION-LEVEL TIMING MODEL [5].

	Sobel filter dataset			Gaussian filter dataset		
	WILD	existing	reduction	**WILD**	existing	reduction
Int_Add	521	826	33%	502	897	44%
Int_Mul	1370	2187	37%	1654	2241	26%
FP_Mul	2112	2438	13%	2057	2482	17%

- False positive: this will still ensure the circuit safety but incurs performance penalty because it uses larger clock period than needed.
- False negative: this will result in timing violation in the circuit. We assume the *detection-and-correction* is used here to correct the timing violations by using *instruction replay* [6], which will flush the pipeline.

We calculate the average timing delay of an instruction as:

$$average\_delay_{inst} = \frac{total\_running\_time_{inst}}{\#inst\_running} \quad (4)$$

where the $total\_running\_time_{inst}$ is the sum of running clock period of each instruction instance in the application execution, including the false positive and false negative induced cycle penalty, and $\#inst\_running$ is the total number of instruction instances executed in the application. We compare the **WILD**-directed DFS with the DFS directed by existing instruction-level model [5] in Table. IV. We observe that by using **WILD** directed DFS, the average instruction-level timing delay can be reduced 13–44% compared to the existing model. This timing margin reduction can be further utilized online with DFS to accelerate the program execution.

V. DISCUSSION

Implementation of DFS: Dynamic frequency scaling (DFS) has been used to adjust operating frequency according to real-time circuit delay during runtime to improve performance [5] [21] [20], implemented by a CGU. CGU is a fast adaptive clock controller circuit that can be designed using a couple of PLLs, each of which is running at a fixed frequency independently, and a multiplexer is used to select one specific frequency within a single cycle [23]. Moreover, a compact all-digital phase-locked loop (ADPLL) clock generator has been proposed in [10] with a ultra-low overhead in power and area, which can provide frequency switching arbitrarily in a wide range within a single clock cycle. However in reality, the frequency can only scale in discrete steps, so the operating frequency can only be selected within a fixed number of options. In this paper, we consider five different classes of dynamic delay of a circuit to adjust the operating frequency, which can then be implemented by a 5-PLL CGU design. More details of DFS implementation can refer to [21] [23] [28].

Overhead of WILD: Although the on-chip model can aid the DFS to achieve better performance, the hardware overhead of implementing the learning model does need special care. An on-chip Gaussian-kernel based SVM is proposed using an analog circuit [15]. Recently, a voltage-droop induced circuit delay prediction model has been implemented using SVM to augment online DFS, whose hardware overhead is 1.5% for today's processor design [28]. We expect the overhead of **WILD** model is less than SVM since LR is less complex than SVM. In our experiment, the SVM classification time is more than 100X of LR model. Besides, the **WILD** model runs remarkably faster than gate-level simulation. To compute 200k test data for dynamic delay, **WILD** is 60X faster than gate-level simulation. Our future work focuses on developing machine learning model that has more efficient computing time and higher prediction accuracy. We also look forward to implementing learning model with a low-overhead hardware implementation with emerging technology.

VI. RELATED WORK

Various techniques have been proposed to eliminate the pessimistic timing margin while protecting from errors. They are classified as follows.

Correcting Errors: *BTWC* approach typically performs frequency overscaling operations which allow timing error occurrences and then recover them by masking timing errors. This kind of technique relies on timing error detection module, usually accomplished by a shadow latch [9] and timing error correction module, which can be realized by instruction replay [6] or global clock gating [9]. Although the *BTWC* approaches are proven to be effective in reducing guardband, the silicon monitoring and error recovery might incur high overhead, especially in case of high timing error rates in the system. For example, [4] has shown that the penalty cost could be 3*N recovery cycles per error, where N is the number of pipeline stages.

Predicting Errors: Therefore, several less-intrusive efforts

were proposed to predict the timing delay in advance and then preventing it by adjusting operating frequency. A program counter (PC)-based prediction scheme has been proposed to identify critical instructions during runtime [22]. Upon the identification of critical instructions, the pipeline is stalled one or two cycles to allow enough timing margin to finish the instruction execution. However, they are not able to consume all available timing slack as they lack in knowledge of absolute value of instruction timing requirements. A recent work quantifies the dynamic timing delay of instructions and enable an instruction-based dynamic frequency scaling (DFS) during runtime to improve performance [5]. However, all these works overlooks the effect of input operands. Until recently, a bit-level timing error prediction model has been proposed to predict the whether a bit position is erroneous or not under given input operands. However, the targeted granularity on bit-level timing error rates limits its usage [13].

Learning in Circuits: Machine learning has been used to model the behavior of circuits under different scenarios. A bayesian network-based model has been used to analyze the reliability of PLC system [12] [11]. A support vector machine (SVM)-based model to predict the circuit delay under voltage droop is proposed in [28]. It also presents the low-overhead hardware design for the learning module that can be applied to any digital block/core, which can augment and strength the utilization of our model on different scenarios. Analog circuits have used machine learning to represent its performance. For instance, the feasibility of output values are predicted using SVM given a set of input parameters [7]. *In our work, we extend the utilization of machine learning to digital circuits timing modeling.*

VII. Conclusions

WILD is a workload-based supervised learning model to predict the dynamic delay of functional units for a given input workload. Its need is driven by growing pressure to reduce timing margins in traditional design. To calibrate its effectiveness, we perform dynamic timing analysis on a post-layout netlist and extract useful 'features' that affect the circuit-level dynamic path sensitization, and hence the dynamic delay. The model is trained using logistic regression with training data from random generation and application profiling, and tested using unseen data from three different datasets. Across three functional unit types and three datasets, **WILD** exhibits high prediction accuracy up to 99.8% (average 98.0%) and achieves a prediction accuracy of 3.6X and 1.5X higher compared to two baseline models. Compared with the state-of-art instruction-level timing model enabled dynamic frequency scaling (DFS), with **WILD**-directed DFS, the average instruction-level timing delay could be reduced by 13%–44% depending on the type of instruction and dataset.

VIII. Acknowledgments

The authors would like to thank Professor Andreas Burg of EPFL for valuable discussions regarding dynamic delay measurement. This material is based upon the work supported by the National Science Foundations Variability Expedition in Computing under Award No. 1029783. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Amd app sdk v2.5. [online]. available: http://www.amd.com/stream.
[2] Todd Austin et al. Opportunities and challenges for better than worst-case design. In *ASP-DAC*. ACM, 2005.
[3] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.
[4] Keith Bowman et al. A 45 nm resilient microprocessor core for dynamic variation tolerance. *JSSC*, 2011.
[5] Jeremy Constantin et al. Exploiting dynamic timing margins in microprocessors for frequency-over-scaling with instruction-based clock adjustment. In *DATE*. EDA Consortium, 2015.
[6] Shidhartha Das et al. Razorii: In situ error detection and correction for pvt and ser tolerance. *JSSC*, 2009.
[7] Fernando De Bernardinis et al. Support vector machines for analog circuit performance representation. In *DAC*. IEEE, 2003.
[8] Florent De Dinechin et al. Designing custom arithmetic data paths with flopoco. *IEEE Design & Test of Computers*, (4):18–27, 2011.
[9] Dan Ernst et al. Razor: A low-power pipeline based on circuit-level timing speculation. In *MICRO-36.*, 2003.
[10] Sebastian Hoppner et al. A compact clock generator for heterogeneous gals mpsocs in 65-nm cmos technology. *TVLSI*, 2013.
[11] Yu Jiang et al. Bayesian-network-based reliability analysis of plc systems. *IEEE transactions on industrial electronics*, 2013.
[12] Yu Jiang et al. System reliability calculation based on the run-time analysis of ladder program. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*. ACM, 2013.
[13] Xun Jiao et al. Supervised learning based model for predicting variability-induced timing errors. In *Proc. of NEWCAS*. IEEE, 2015.
[14] Andrew B Kahng et al. Slack redistribution for graceful degradation under voltage overscaling. In *ASP-DAC*. IEEE, 2010.
[15] Kyunghee Kang et al. An on-chip-trainable gaussian-kernel analog support vector machine. *TCAS I*, 2010.
[16] Michael Kearns et al. Cryptographic limitations on learning boolean formulae and finite automata. *JACM*, 1994.
[17] Michael Kharitonov. Cryptographic hardness of distribution-specific learning. In *STOC*. ACM, 1993.
[18] Veit B Kleeberger et al. Workload-and instruction-aware timing analysis: The missing link between technology and system-level resilience. In *DAC*. ACM, 2014.
[19] Fabian Pedregosa et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.
[20] Abbas Rahimi et al. Hierarchically focused guardbanding: an adaptive approach to mitigate pvt variations and aging. In *DATE*. IEEE, 2013.
[21] Abbas Rahimi et al. Application-adaptive guardbanding to mitigate static and dynamic variability. *Computers, IEEE Transactions on*, 2014.
[22] Sanghamitra Roy et al. Predicting timing violations through instruction-level path sensitization analysis. In *DAC*. ACM, 2012.
[23] James Tschanz et al. Adaptive frequency and biasing techniques for tolerance to dynamic temperature-voltage variations and aging. In *ISSCC*. IEEE, 2007.
[24] James Tschanz et al. Tunable replica circuits and adaptive voltage-frequency techniques for dynamic voltage, temperature, and aging variation tolerance. In *2009 Symposium on VLSI Circuits*, 2009.
[25] G Tziantzioulis et al. b-hive: a bit-level history-based error model with value correlation for voltage-scaled integer and floating point units. In *DAC*. ACM, 2015.
[26] Rafael Ubal et al. Multi2Sim: A Simulation Framework for CPU-GPU Computing . In *Proc. of PACT*, Sep. 2012.
[27] Peter Welinder, Steve Branson, Takeshi Mita, Catherine Wah, Florian Schroff, Serge Belongie, and Pietro Perona. Caltech-ucsd birds 200. 2010.
[28] Fangming Ye et al. On-chip voltage-droop prediction based on support-vector machines and feature selection. *TCAD*, 2016.