Improving Test Chip Design Efficiency via Machine Learning*

Zeye Liu, Qicheng Huang, Chenlei Fang and R. D. (Shawn) Blanton
Advanced Chip Testing Laboratory (www.ece.cmu.edu/~actl/)
Department of Electrical and Computer Engineering
Carnegie Mellon University

***Abstract*—Competitive position in the semiconductor field depends on yield which is becoming more challenging to achieve high levels due to the increasing complexity associated with the design and fabrication of leading-edge integrated circuits (ICs). Consequently, test chips, especially full-flow logic test chips, are increasingly employed to investigate the complex interaction between layout features and the process before and during product ramp. However, designing a high quality full-flow logic test chip can be time-consuming due to the huge design space. This work describes a design methodology that deploys a random forest classification technique to predict synthesis outcomes for test chip design exploration. Experiments on creating five full-flow logic test chips, which mimic five different designs, demonstrate the efficacy of the proposed methodology. To be specific, those design experiments demonstrate that the machine learning aided flow speeds up design by 11× with negligible performance degradation.**

I. Introduction

The continued scaling of integrated circuits (ICs) has made the semiconductor industry extremely capital intensive. Sub-wavelength lithography and local layout effects create layout dependencies at the 16nm node and below, which makes fast yield ramping increasingly challenging [1]. Thus, there can be substantial economic benefits for fast yield ramping for a leading-edge technology node. In other words, aggressive yield loss reduction is crucial for successful IC manufacturing [2].

Conventionally, various test chips are used at different technology development stages, with respect to the different stages of fabrication maturity [3]. Fig. **1** describes a six-stage development flow for a generic technology, along with the types of test chips manufactured at each stage. At the initial stage, simple proof-of-concept structures, such as comb drives and via arrays [4], are used to evaluate fabrication steps independently. As the process development proceeds, more complicated test chips are produced, such as SRAM blocks and short-flow test chips with either FEOL or BEOL layout characteristics [5]. As the process defectivity reduces, full-flow logic (FFL) test chips are manufactured [6] using the developing product design kit (PDK). FFL test chips refers to standard automated place-and-route (SAPR) logic test chips that are intended to identify significant sources of yield loss that impact the random logic within product design. In addition, since the full-flow SAPR logic test chip follows a standard flow, they can be used by fabless companies to identify and mitigate the product yield losses before high-volume product manufacturing begins.

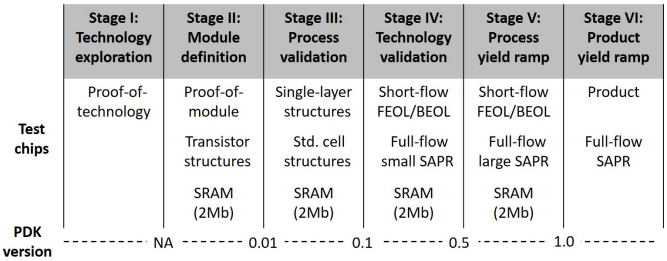

Figure 1. Outline of a generic semiconductor technology development process.

The most common FFL test chips employed in industry are sub-circuits (e.g., a floating-point unit) from existing product designs. While such sub-circuits contain actual design features (e.g., standard cell usage and complex layout geometries), the primary drawback is the low transparency to a large universe of failures, which results in difficult failure analysis (yield learning) for conventional FFL test chips. To address this shortcoming, work in [7][8] describes a new type of FFL test chip called the Carnegie-Mellon Logic Characterization Vehicle (CM-LCV). The CM-LCV is designed for maximal testability and diagnosibility while being sensitive to the defect mechanisms that affect product designs. It is based on the insight that systematic defects are sensitive to the physical features of a design (i.e., layout geometries) instead of the logic functionality. This provides the freedom to select a logical functionality and structure that maximizes testability and diagnosability, and a layout implementation that has product-like physical features (e.g., standard cell usage and complex layout geometries). Particularly, the CM-LCV is a two-dimensional array of functional unit blocks (FUBs) that implement one or more information-lossless functions with equal numbers of inputs and outputs. Both the structure and functionality of the CM-LCV maximizes testability and diagnosability for a variety of defect types [7].

One important step of the CM-LCV design flow is the creation of a FUB library that includes various *unique* FUB implementations. "Unique" here means the logical structure of a FUB implementation is different from any other implementation within the FUB library. Having a FUB library full of unique FUBs means design reflection objectives such as matching standard cell usage is eased. In other words, it is both easier and more likely to identify a set of unique FUB implementations such that the distribution of standard cells within the set matches a targeted design. Note that matching

*This research is sponsored by the Semiconductor Research Corporation (SRC) [Task id: 2785.001].

978-1-7281-4823-6/19/$31.00 ©2019 IEEE

	NAND2	AND2	OR2	OR3	XOR2	XOR3
Target distribution 1	9	3	0	1	0	1
Target distribution 2	9	11	4	1	8	1
FA1	9	0	0	0	0	0
FA2	0	3	0	1	0	1
FA3	0	2	1	0	2	0

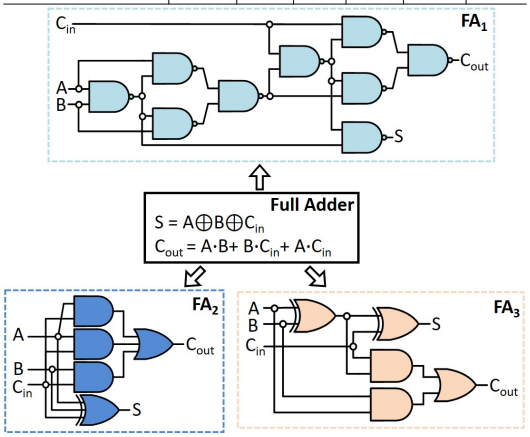

Figure 2. An example of matching two target standard-cell distributions with unique full adder implementations.

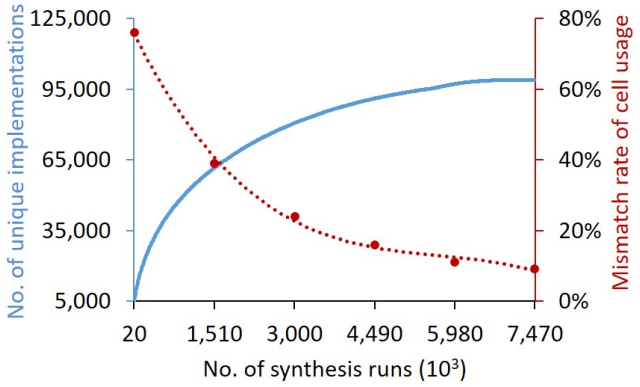

Figure 3. The number of unique FUB implementation (in solid line) and mismatch rate of cell usage (in dash line) as a function of the number of synthesis runs. The dash line is fitted by six samples, where six CM-LCVs are created to measure the mismatch.

the standard cell usage is crucial to test chip design since it enables the investigation of intra-cell layout geometries. Fig. **2** illustrates simple examples of matching two targets with several unique full adder (FA) implementations FA_1, FA_2 and FA_3. The first target distribution includes four standard-cell types (i.e., NAND2, AND2, OR3, and XOR3). Two implementations (FA_1 and FA_2) match the first distribution perfectly. The second distribution includes two new cell types (OR2 and XOR2), a distribution which cannot be matched using only FA_1 and FA_2. However, using implementation FA_3, we can perfectly match the second distribution using one instance of FA_1, one instance of FA_2, and four instances of FA_3. The CM-LCV does not use the adder function for a number of reasons but the example, however, illustrates that a variety of FUB implementations is crucial for matching a given standard-cell distribution, and additionally, to achieve flexible physical feature incorporation.

Unfortunately, creating a variety of unique FUB implementations is extremely expensive, since a FUB function has to be synthesized millions of times to accurately meet the design requirements. The reason that significant synthesis is needed is due to the fact that synthesis cannot guarantee the generation of a new unique FUB implementation that is different from all those previously generated. For each synthesis run, although a new configuration (e.g., requirements of which standard cells can be used; more details in Section II) is provided as input, the result can still be a non-unique implementation that satisfies the configuration. Fig. **3** gives some statistics collected from a FUB library for previous CM-LCVs that required six weeks of generation time using 64 2.2GHz CPU cores and 1TB of RAM. The solid line shows that the number of unique implementations grows slowly with more synthesis. On average, 75 synthesis runs are required to produce one unique implementation. Overall, nearly 7.5 million synthesis runs are performed in order to reduce the *mismatch rate of cell usage** (as shown in the dash line) to an acceptable level. Therefore, if it can be learned which synthesis configurations are likely to lead to unique implementations before synthesis is executed, then we can avoid many useless runs, and the efficiency of the CM-LCV design process can be significantly improved.

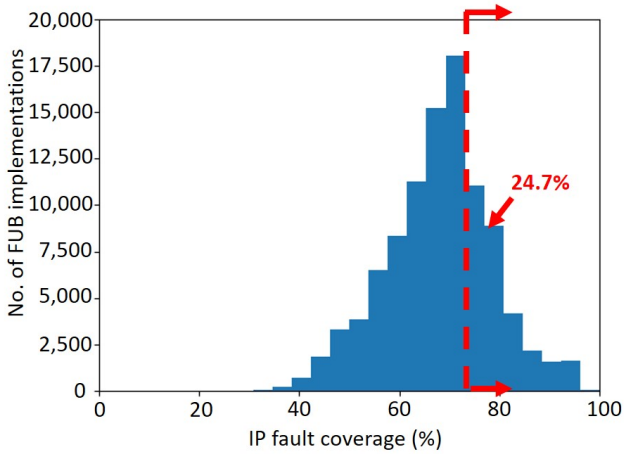

Figure 4. IP fault coverage distribution of various FUB implementations.

Another challenge of FUB library creation is to ensure high testability of the logic-level implementations. Since one objective of the CM-LCV is to achieve high transparency to defects, after the FUB library is established, the testabiliy of each unique FUB implementation has to be measured by an automatic test pattern generator (ATPG) to obtain coverage for various fault models of concern. FUB implementations with low testability are disqualified from use within the CM-LCV, and thus waste significant compute resources when measuring their poor testability. Fig. **4** shows coverage distribution of

*Mismatch rate of cell usage is the error between the target cell usage distribution and the one of CM-LCV. Particularly, it is calculated as $\frac{\sum|\Delta_i|}{T}$, where $|\Delta_i|$ is the absolute difference between the number of instances of standard-cell i in the design and the CM-LCV, and T is the total number of cells in the target design.

the input pattern (IP) fault model [9] for all the unique FUB implementations created in the process illustrated in Fig. **3**. Because the IP fault model ensures the detection of every irredundant intra-cell defect, it is adopted here to gauge the cell-level testability of the CM-LCV. From the distribution, we observe that only 24.7% of the unique implementations achieve at least 75% IP fault coverage. Therefore, significant resources can be saved if the testability for each synthesized FUB implementation can be accurately predicated as satisfactory or not.

The aforementioned observations motivate exploration of synthesis/ATPG reduction by predicting the outcomes of synthesis and ATPG to accelerate the CM-LCV design process. Particularly, the objective of this work is to develop a methodology to predict whether a certain operation (i.e., synthesis with a certain configuration or testability measurement on a certain FUB implementation) will result in a satisfactory outcome (i.e., a unique or highly-testable FUB implementation, respectively). Preliminary analysis (details are given in Section II) reveals that there is no simple correlation either between synthesis configuration and uniqueness of the synthesis outcome, or between the circuit structure and testability. Based on the recent successes of machine learning (ML) in uncovering higher-dimensional correlations [10][11], we deploy random forests (RFs) for predicting FUB uniqueness and FUB testability. The contributions of this work include:

- Development of two RF classifiers to predict (i) whether a synthesis configuration will result in a unique FUB implementation, and (ii) whether a FUB implementation has an acceptable testability, so as to avoid unnecessary synthesis and testabiliy analysis.
- Development of an on-line learning strategy for cases without sufficient training data.
- Design experiments to create various CM-LCVs using the flow in [12] and the new ML-aided flow, which demonstrates the latter achieves up to 11× speed-up with negligible performance degradation.
- While this methodology is proposed specifically for CM-LCV design, it can also be used for the design of other FFL test chips that involve synthesis and testability analysis. The developed RF models can also be generalized for other synthesis or testability applications.

The rest of the paper is organized as follows. Section II provides the relevant background for FUB library creation. Section III describes the proposed ML methodology to accelerate the design process for CM-LCVs. Experiment results assessing and validating the methodology are presented in Section IV. The final section summarizes the paper.

II. Background

In this section, we introduce a typical design flow of CM-LCV and the details of synthesis configuration to shed light on (i) how this work contributes to the overall acceleration of CM-LCV design, and (ii) why there is no simple correlation between synthesis configuration and the prediction goals, thus motivating the need for ML deployment.

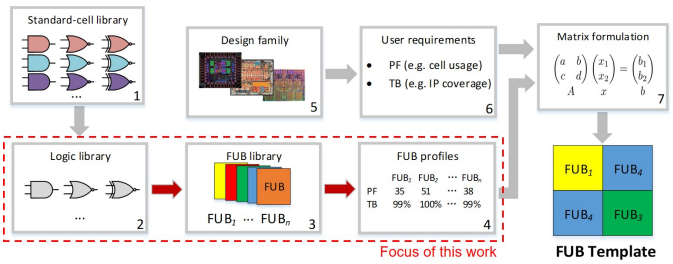

Figure 5. The CM-LCV design flow, steps in dashed box indicate bottlenecks that are optimized in this work.

Fig. **5** illustrates the CM-LCV design flow described in [12-14], in which CM-LCVs are designed to match standard-cell distributions [12], mimic the neighborhood of standard cells [13], and incorporate BEOL layout geometries of interest [14]. The design flow begins with the standard-cell library for a given technology, as shown in Step 1 of Fig. **5**. A typical standard-cell library contains numerous logic functions with different drive strengths. Logic functions with different drive strengths are indicated by different colors within each column in Step 1 of Fig. **5**. For example, there may exist a two-input AND with nominal drive strength (typically denoted as and2×1) and a two-input AND with twice the drive strength (e.g., and2×2). Both cells implement the same two-input AND logic function. Logic functions are extracted from the standard-cell library to create a logic library, as shown in Step 2 of Fig. **5**. Based on the logic library and a FUB function, Step 3 of Fig. **5** generates a large variety of unique FUB implementations using the logic functions extracted from the library. This step is achieved conventionally through millions of synthesis runs with different configurations with retainment of the unique implementations which then constitute the FUB library. Each FUB is analyzed to determine its physical features (PF) and testability characteristics (TB), resulting in a set of profiles (Step 4 of Fig. **5**).

On the other hand, the design objectives of the CM-LCV usually include testability requirements (e.g., IP coverage) and objective physical features (e.g., standard-cell distribution). The latter can be taken from representative industrial designs (Step 5 of Fig. **5**) or be directly specified by the designers (Step 6 of Fig. **5**). Given the design requirements and the FUB library along with their profiles, the ultimate goal of the design flow is to identify a subset of FUB implementations that satisfy the design requirements. Identifying a subset of FUBs can be achieved by solving an optimization problem, as illustrated in Step 7 of Fig. **5**. The solution indicates which

	K-2 logic functions				Performance goal
	AND2	XOR2	…	AOI22	
$\mathbf{x}^{(1)}$	1	0	…	0	1
$\mathbf{x}^{(2)}$	0	1	…	0	1
…	…	…	…	…	…
$\mathbf{x}^{(n)}$	1	1	…	1	3

Figure 6. Illustration of n synthesis configurations, where each is represented by a vector.

implementations and how many instances should be included to form what we call the FUB template. For example, the final FUB template in Fig. **5** consists of one FUB_2, one FUB_3 and two instances of FUB_4.

In spite of the numerous steps of the design flow, the main bottleneck lies in the two steps indicated by the red arrows, namely, synthesis for generating a variety of unique FUB implementations, and the FUB analysis to measure testability for each implementation. Our work aims to accelerate these two bottle-neck steps, which can be more than 20× costly in runtime as compared to the other steps in Fig. **5**.

The reason that FUB library generation is so time consuming is that millions of synthesis runs have to be executed with different configurations to generate a sufficient number of unique FUB implementations. Fig. **6** illustrates an example of n synthesis configurations, where each configuration is represented by a row-vector. A synthesis configuration includes two parts: (i) a binary vector that constrain which logic gates can be used in the resulting implementation, as shown in blue, and (ii) a goal performance metric indicator, as shown in orange. In the first part, a "1" and a "0" indicate which gates are allowed and not allowed, respectively. The length of the binary vector is $K-2$, where K is the number of logic functions in the logic library, because primitive logic functions such as the 2-input NOR and inverter are always required by the synthesis tool [15]. The second part, namely the goal performance metric indicator, is a one-digit integer indicating which performance goal that the synthesis should attempt to achieve with priority. Three goals are used in [12-14]: minimal area, minimal delay and balanced delay and area, indicated by "1" to "3", respectively. For example, the vector $\mathbf{x}^{(1)}$ in Fig. **6** implies a synthesis configuration that is expected to generate a FUB implementation that only uses the logic functions consisting of 2-input AND, 2-input NOR and inverter, while minimizing the overall circuit area. Note that any change to the vector or the performance goal leads to a new synthesis configuration, but does not necessarily result in a unique FUB implementation, because synthesis does not necessarily use all the specified logic functions.

Based on the aforementioned, the number of possible synthesis configurations for a FUB function is estimated to be:

$$D = (2^{(K-2)} - 1) \times g, \tag{1}$$

where K is the number of logic functions in the logic library, and g is the number of performance metrics. The space for all the possible configurations is extremely large, and thus too time-consuming to exhaustively explore. For example, the standard-cell library used in the example of Fig. **3** has 58 logic functions, which translates to 2.9×10^{17} synthesis configurations when there are $g = 3$ possible performance metrics. Without ML, an extensive amount of synthesis is required to generate a sufficient number of unique FUB implementations. If instead we can predict which configurations will lead to unique implementations before synthesis, useless synthesis runs can be avoided in order to save significant compute time.

However, predicting the outcome of synthesis is not trivial, especially given the high dimensionality of a configuration vector. In order words, it is difficult to uncover the correlation between high-dimensional features and the objective, either from experience or through simple model fitting (e.g., polynomials). The same challenge exists for predicting the testability of FUB implementations in order to accelerate FUB analysis. Therefore, we use ML techniques to learn the complex correlations within the high-dimensional space.

III. Design Methodology

In this section, we describe the details of the proposed methodology for efficient CM-LCV design. We first introduce the new design flow and formulate the corresponding mathematical problems. Then we describe the deployed ML algorithm and the features used for learning. Finally, we illustrate an on-line learning strategy, which is used to train the ML model starting from limited labeled data.

A. Design Flow and Problem Formulation

Fig. **7** shows the updated design steps (Steps 2 to 4 of Fig. **5**). The steps in gray are an exact copy of the steps from the conventional flow in Fig. **5**, and the two bottle-neck processes described in Section II (i.e., synthesis and ATPG for testability measurement) are indicated by red. The flow charts in blue illustrate the proposed steps to accelerate the two bottle-neck processes. Instead of inputting the configurations randomly into the synthesis tool or feeding the entire FUB library into the ATPG tool, the two loops in blue act as two filters that only select (i) configurations predicted to be unique, and (ii) FUB implementations that are predicted to have high testability. The two filters are correspond to two classifiers C_1 and C_2.

Here we choose to train two classifiers instead of just one that predicts the testability of a synthesized FUB directly from the configuration. That is because the testability of a FUB implementation is highly associated with its characteristics (e.g., circuit structure, number of fanouts per net, etc.) Without such characteristics, a very complex model (e.g., deep neural networks) is needed to represent the extremely complicated correlation between synthesis configuration and the resulting testability. The amount of time and data for training such a highly-complex model is simply too significant for the task at hand. Instead, separating the difficult task into two stages and using two relatively simple models such as RFs is fast and results in good performance through on-line learning with a small amount of data.

Corresponding to the two objectives (i.e., predicting uniqueness of the synthesis outcome, and FUB testability), our work aims to solve the following two sub-problems:

- **Problem 1: uniqueness prediction**
  Suppose we have n possible synthesis configurations and for each of them d features are extracted. Such data constitute the testing set, which can be represented as an $n \times d$ matrix $\mathbf{X} = \left[\mathbf{x}^{(1)}; \mathbf{x}^{(2)}; \ldots; \mathbf{x}^{(n)}\right]$. Each row

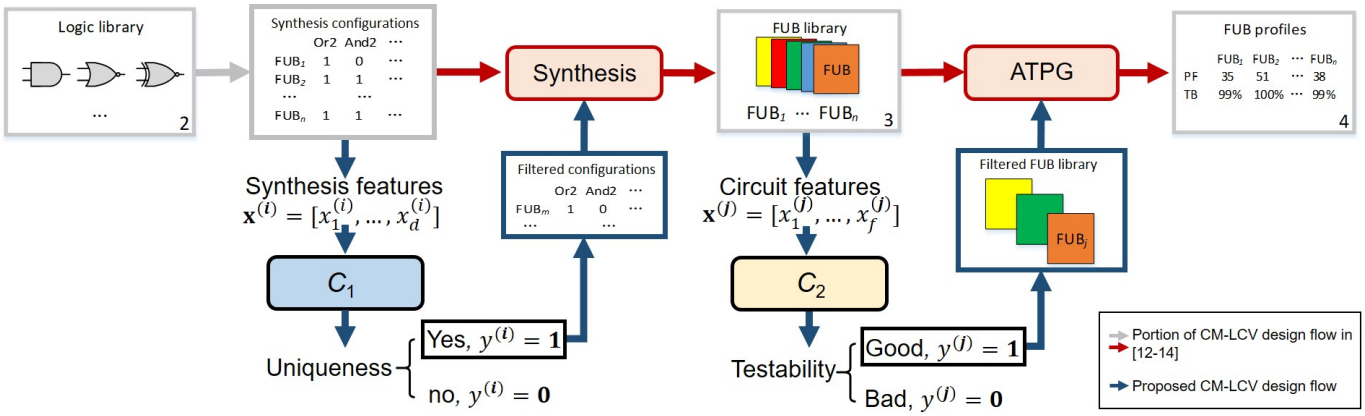

Figure 7. The use of RF models in the original CM-LCV design flow: a classifier C_1 is trained using features derived from a synthesis configuration to predict synthesis outcome; and a classifier C_2 is trained using features derived from a unique implementation to predict its testability.

$\mathbf{x}^{(i)} = \left[x_1^{(i)}, x_2^{(i)}, \ldots, x_d^{(i)}\right]$ is a d-dimensional vector containing the d extracted features of the i-th configuration. The objective of this sub-problem is to train a classification model C_1, which takes in each test sample $\mathbf{x}^{(i)}$ and generates a label $y^{(i)}$ representing the uniqueness of the corresponding synthesized FUB implementation. $y^{(i)}$ is a binary variable, which equals one when the implementation resulting from the i-th synthesis configuration is unique (i.e., distinguishable from any existing implementations within the FUB library), and equals zero otherwise. For an optimal C_1, the predicted labels $\mathbf{y} = \left[y^{(1)}, y^{(2)}, \ldots, y^{(n)}\right]$ should be as close to the real labels as possible.

- **Problem 2: testability prediction**
  After synthesis, suppose there are m unique implementations selected into the FUB library. f features are extracted from each implementation, which form the test set. The test set can be represented as an $m \times f$ matrix $\mathbf{X} = \left[\mathbf{x}^{(1)}; \mathbf{x}^{(2)}; \ldots; \mathbf{x}^{(m)}\right]$, where each row $\mathbf{x}^{(i)} = \left[x_1^{(i)}, x_2^{(i)}, \ldots, x_f^{(i)}\right]$ is a f-dimensional vector containing the f extracted features of the i-th FUB implementation. Given a threshold for acceptable testablilty, the objective of this sub-problem is to train a classification model C_2, which takes in each test sample $\mathbf{x}^{(i)}$ and generates a label $y^{(i)}$ representing whether the testability of the FUB implementation is acceptable. $y^{(i)}$ is again a binary variable that equals one when the testabilility of the i-th FUB implementation is larger than the pre-defined threshold, and equals zero if not. For an optimal C_2, the predicted labels $\mathbf{y} = \left[y^{(1)}, y^{(2)}, \ldots, y^{(m)}\right]$ should be as close to the real labels as possible.

B. Feature Selection

To achieve optimal classification performance, the features should be carefully selected to best represent the raw data, and also incorporate helpful domain knowledge.

The feature selection for classifier C_1 is straightforward. Because a compacted vector, illustrate in Fig. **6**, already includes all the information of a synthesis configuration, it can be directly used as the feature vector. To be specific, given a logic library of K logic functions, $K-1$ features are extracted from each configuration, where the first $K-2$ features are binary numbers representing whether a certain logic function (excluding NOR2 and inverter) in the logic library is allowed for synthesis, and the last feature is a one-digit variable that captures the performance goal for synthesis (as explained in Section II).

	K logic functions				No. of nets	No. of fanout	Max logic depth
	INV	NOR2	…	XOR3			
$\mathbf{x}^{(1)}$	12	2	…	0	25	36	9
$\mathbf{x}^{(2)}$	1	8	…	3	51	68	6
…	…	…	…	…	…	…	
$\mathbf{x}^{(m)}$	3	2	…	1	36	58	5

Figure 8. Illustration of selected features for m unique implementations for classifier C_2.

For classifier C_2, the raw data includes implementations in the form of gate-level netlist. Unlike C_1's case where the synthesis configuration can be directly converted into a vector, for C_2, features must be manually designed with the cost of extraction kept in mind. Circuit testability largely depends on its topology, so we extract $K+3$ features to represent the topology and connections. Fig. **8** illustrates an example of the $K+3$ features for m implementations. The first K features captures the cells utilized in a FUB implementation. For example, the feature vector $\mathbf{x}^{(1)}$ shown in Fig. **8** is for a FUB with 12 inverters, two 2-input NOR gates, etc. In addition, we include three more features to capture information concerning circuit structure, namely, the number of nets, the number of fanouts, and the maximum logic depth. We do not consider other features associated with testability because extraction is too expensive, such as the number of re-convergent fanouts. The time required to extract such features exceeds the time needed for ATPG, thus making the use of those features

nonsensical.

C. Classification Algorithm

Among the variety of available ML algorithms, the RF has become popular because it has good performance and is easy to implement. In addition, the RF is capable of classifying nonlinearly separable data with a short learning time. Given those aforementioned advantages, we use RF models to construct predictors C_1 and C_2.

An RF is an ensemble method based on decision trees. A decision tree learns a tree-structured model from the training samples, with each leaf representing a classification result. For each internal node of the tree, one feature is selected as the optimal split criteria at the current level. While a decision tree is easy to implement and interpret, it is not robust. A small change in the training samples can result in a totally different tree. A single decision tree is also prone to over-fitting the training set. In addition to a binary label of either "0" or "1", a decision tree can also provide a probabilistic label, which is a probability for the label to be "1". Such a probability is calculated from the ratio of label-1 testing instances within the leaf node.

An RF overcomes the disadvantages with ensemble learning [16]. An RF is an ensemble of decision trees with two degrees of randomness. First, the training samples for each tree is generated from bootstrap sampling (random sampling with replacement) of the entire training set. In this way, each tree has a different set of training data, although drawn from the same distribution. Second, when searching for the optimal split at each node, only a subset of all features are selected. A random forest model performs final classification by taking a majority vote over the trees. From these two degrees of randomness, an RF model achieves much lower variance than a single decision tree, at the expense of slightly increasing the fitting bias. Note that an RF can also produce a probabilistic label, which is the average of the probabilistic labels predicted by all the decision trees.

D. On-line Learning Strategy

An on-line learning strategy is developed specifically to tackle the insufficient-data problem faced in the training process of RFs. Since there is no quicker way to obtain the labels for the training set other than running synthesis/ATPG, we want to minimize the size of the training set to minimize cost. However, a small training set can easily lead to under-fitting the RF models which results in greater levels of misprediction.

On-line learning involves iteratively updating the ML model as more training data becomes available. It starts with an insufficient training set, so a model initially learned may be far from optimal. However, as more training data are gradually added, the updated model becomes more accurate compared to the previous versions. In our work, new training data stems from verification of the prediction results. For C_1, after prediction, the configurations predicted with label "1" are synthesized to generate implementations. Synthesis not only provides the implementations for the next design stage, but also ground-truth labels after circuit structure comparison. Such data can augment the training set for an ML model update. For C_2, similarly, the implementations predicted with label "1" are analyzed for testability via ATPG. This process provides ground-truth labels while also verifying the predicted testability.

Two requirements are necessary for an efficient on-line learning process: (i) fast model training in each iteration, and (ii) inexpensive creation of additional training data. Without either requirement, model update would be too costly. Fortunately, classifier C_1 and C_2 satisfy both requirements. Specifically, training time for an RF is negligible (usually less than one minute) compared to other operations such as synthesis. In addition, modest effort is needed for obtaining the labels for new training data. For C_1, the synthesis process cannot be skipped since the resulting implementations are eventually required for creating the FUB library. The additional effort is simply circuit structure inspection for identifying uniqueness. Similarly, for C_2, the additional effort requires ATPG to determine if coverage exceeds the pre-defined threshold.

Algorithm 1 On-line Learning Strategy for classifier C_1

Input: An unlabeled dataset $\mathbf{X}$
Output: Predicted labels $\mathbf{Y}$ for the input dataset $\mathbf{X}$
//Initialization stage
1. $\mathbf{X}_{\text{init}} \leftarrow N$ random samples from $\mathbf{X}$
2. $\mathbf{Y}_{\text{top}} = \mathbf{Y}_{\text{init}} = \text{get\_label\_by\_syn}(\mathbf{X}_{\text{init}})$
3. $\mathbf{D}_{\text{train}} = \{\mathbf{X}_{\text{init}}, \mathbf{Y}_{\text{init}}\}$
4. $\mathbf{X}_{\text{test}} = \{\mathbf{X} - \mathbf{X}_{\text{init}}\}$
//On-line learning stage
while (1s count in $\mathbf{Y}_{\text{top}} \geq 1$) **do**
    5. Train C_1 with $\mathbf{D}_{\text{train}}$
    6. $\mathbf{P}_{\text{test}} = C_1(\mathbf{X}_{\text{test}})$
    7. $\mathbf{X}_{\text{top}} = \text{top\_ranked}(\mathbf{P}_{\text{test}}, \mathbf{X}_{\text{test}})$
    8. $\mathbf{Y}_{\text{top}} = \text{get\_label\_syn}(\mathbf{X}_{\text{top}})$
    9. $\mathbf{X}_{\text{btm}} = \text{bottom\_ranked}\ (\mathbf{P}_{\text{test}}, \mathbf{X}_{\text{test}})$
    10. $\mathbf{Y}_{\text{btm}} = \text{get\_label\_by\_asg}(\mathbf{X}_{\text{btm}})$
    11. $\mathbf{X}_{\text{sel}} = \{\mathbf{X}_{\text{top}}, \mathbf{X}_{\text{btm}}\}$; $\mathbf{Y}_{\text{sel}} = \{\mathbf{Y}_{\text{top}}, \mathbf{Y}_{\text{btm}}\}$
    12. Augment $\mathbf{D}_{\text{train}}$ with $\{\mathbf{X}_{\text{sel}}, \mathbf{Y}_{\text{sel}}\}$
    13. Replace $\mathbf{X}_{\text{test}}$ with $\{\mathbf{X}_{\text{test}} - \mathbf{X}_{\text{sel}}\}$
end

Algorithm 1 summarizes the on-line learning strategy for C_1. Given a set of synthesis configurations with features extracted as $\mathbf{X}$, the goal is to iteratively find labels for them until no more configuration with label "1" are found. The first part of Algorithm 1 is to set up the initial training and testing sets. We randomly select a small number of samples from $\mathbf{X}$, denoted as $\mathbf{X}_{\text{init}}$ (line 1). Then, synthesis with configurations corresponding to $\mathbf{X}_{\text{init}}$ is executed to label $\mathbf{X}_{\text{init}}$ (line 2). The labels are denoted as $\mathbf{Y}_{\text{init}}$ and the labeling process is denoted as a function named "get_label_by_syn()". The features (i.e., $\mathbf{X}_{\text{init}}$) and their labels (i.e., $\mathbf{Y}_{\text{init}}$) constitute the initial training set $\mathbf{D}_{\text{train}} = \{\mathbf{X}_{\text{init}}, \mathbf{Y}_{\text{init}}\}$ (line 3). The remaining unlabeled portion of $\mathbf{X}$ forms the initial testing set $\mathbf{X}_{\text{test}}$ for C_1 (line 4).

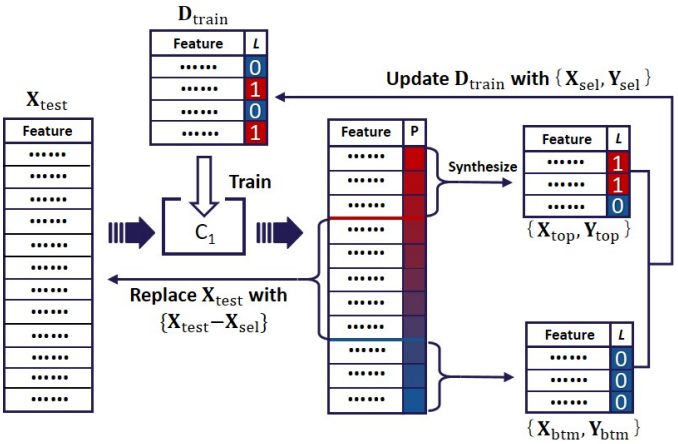

Figure 9. Illustration of one iteration of on-line learning for C_1. After training based on the updated training set of the previous iteration, C_1 predicts a probability for each testing sample to have label "1". The top-ranked samples are synthesized for true labels and the bottom-ranked ones are directly labeled as "0". These two parts of data are added to the training set for next iteration, while the remaining mid-ranked samples are forwarded to the next iteration.

The second part of Algorithm 1 describes on-line learning, where C_1 is updated once during every iteration of the while loop. The update procedure in each iteration is illustrated in Fig. **9**. First, the classifier C_1 predicts a label for all the data samples in the testing set $\mathbf{X}_{\text{test}}$ which results in a vector containing the probabilistic labels $\mathbf{P}_{\text{test}}$ (line 6). Here, a probabilistic label refers to a real number $p \in [0, 1]$, indicating the probability for the sample to be labeled with "1". Then, the samples $\mathbf{X}_{\text{test}}$ are sorted according to their probabilistic labels $\mathbf{P}_{\text{test}}$ from high to low. A fixed number of top-ranked testing samples are selected and denoted as $\mathbf{X}_{\text{top}}$ (line 7). $\mathbf{X}_{\text{top}}$ are those synthesis configurations believed to lead to the unique implementations, and the true labels are actually determined by performing synthesis (line 8).

As a complement of $\mathbf{X}_{\text{top}}$, the bottom samples in $\mathbf{X}_{\text{test}}$ with p less than a threshold (denoted as $\mathbf{X}_{\text{btm}}$ in line 9) are each directly assigned the label of "0". When the threshold is very low (e.g., 0.01), directly assigning a label of "0" (denoted as "get_label_by_asg()" in line 10) is quite accurate. $\mathbf{X}_{\text{btm}}$ with the all-zero labels $\mathbf{Y}_{\text{btm}}$, together with $\mathbf{X}_{\text{top}}$ and the labels $\mathbf{Y}_{\text{top}}$, are added to the training set $\mathbf{D}_{\text{train}}$. Besides increasing the training set $\mathbf{D}_{\text{train}}$, another reason that we include both $\mathbf{X}_{\text{top}}$ and $\mathbf{X}_{\text{btm}}$ is to ensure class balance in $\mathbf{D}_{\text{train}}$ (line 11-12). Finally, the remaining data in $\mathbf{X}_{\text{test}}$ apart from $\mathbf{X}_{\text{top}}$ and $\mathbf{X}_{\text{btm}}$ are used as testing data for the next iteration (line 13). On-line learning continues in this way until few predictions in $\mathbf{Y}_{\text{top}}$ are "1".

Similar to the classifier C_1, on-line learning for classifier C_2 can be performed. The on-line learning flow for C_2 is almost identical to Algorithm 1. The major difference is that instead of synthesizing the configurations to obtain the ground truth for the selected synthesis configurations $\{\mathbf{X}_{\text{top}}\}$, ATPG tools are used to obtain the ground truth by characterizing the testability for the selected FUB implementations.

TABLE I
CHARACTERISTICS OF THE TWO COMMERCIAL STANDARD-CELL LIBRARIES.

Standard-cell library	No. of standard cells	No. of logic functions	No. of possible synthesis configurations
Lib0	7,485	58	2.9×10^{17}
Lib1	11,981	63	9.0×10^{18}

IV. EXPERIMENTS

In this section, we describe the details of design experiments for various CM-LCVs based on two industrial standard-cell libraries. To evaluate the efficacy of the ML-aided methodology, we compare the design effort and characteristics of the CM-LCVs created by (i) the ML-aided flow and (ii) the conventional flow described in [12].

A. Setup

For experiments, we have two standard-cell libraries (Lib0 and Lib1) and five objective standard-cell distributions (three corresponding to Lib0 and two for Lib1) provided by an industrial partner. Characteristics of the two libraries are listed in Table I. For each standard-cell library, two FUB functions are deployed, named as "Cygnus" and "Hercules". So specifically, the CM-LCV design tasks are:
(i) Using Lib0, generate highly-testable, unique implementations for FUB functions Cygnus and Hercules to form a FUB library that is sufficient for creating LCVs that have cell distributions that match to industrial design blocks, named "BlockA" and "BlockB";
(ii) Repeat the first task using instead Lib1 and industrial designs "BlockC", "BlockD" and "BlockE".

Although the ML-aided flow focuses on mitigating the bottlenecks of the conventional flow (the dashed steps of Fig. **5**), in the experiments we execute the entire design flow and use characteristics of the resulting LCV for evaluation. All experiments are completed using a server with 64 2.2GHz CPU cores and 1TB of RAM.

The performance of the ML-aided flow is equated to the individual performances of classifiers C_1 and C_2, as well as the on-line learning strategies shown in Fig. **9**. To evaluate the performance of each classifier, *precision* and *recall* are used to analyze prediction correctness. Precision and recall are calculated based on four statistics: True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN). TP is the number of samples that are truly positive and predicted as positive. FP is the number of samples that are truly negative but predicted as positive. TN and FN are defined similarly. Precision and recall are computed as:

$$Precision = \frac{TP}{TP + FP}$$
$$Recall = \frac{TP}{TP + FN} \quad (2)$$

In Eq. 2, precision represents the probability that a sample predicted positive is truly positive, while recall represents the probability that a truly positive sample is correctly predicted.

TABLE II
PREDICTION OF SYNTHESIS OUTCOME FOR STANDARD-CELL LIBRARY LIB0 AND LIB1.

Library	FUB function	No. of synthesis configurations		No. of unique implementations		Classifier C_1		Synthesis reduction
		Conventional flow	ML-aided flow	Conventional flow	ML-aided flow	Precision	Recall	
Lib0	Cygnus	7,470,000	500,000	99,031	99,026	19.8%	99.9%	14.5×
	Hercules	9,450,000	820,000	313,262	313,255	38.2%	99.9%	11.2×
Lib1	Cygnus	6,750,000	600,000	125,779	125,766	20.9%	99.9%	10.8×
	Hercules	9,240,000	1,010,000	406,849	406,834	40.3%	99.9%	8.9×

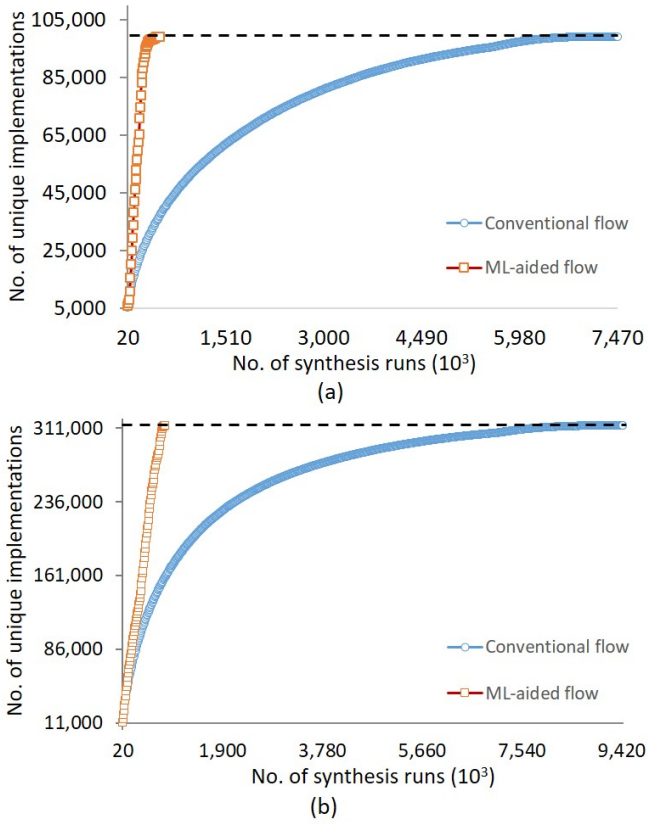

Figure 10. Number of unique implementations generated using standard-cell library Lib0 for FUB functions (a) Cygnus and (b) Hercules.

The precision and recall in Eq. 2 are defined for the positive class, and those for the negative class can be defined similarly. Note that due to the on-line learning strategy, the classifiers can make predictions multiple times. As a result, precision and recall are calculated when on-line learning terminates.

Besides classifier performance, the overall runtime and the resulting LCV desings from the two flows are compared. The classifier performance is examined in Sections IV.A and IV.B, and comparisons of the design flow is accomplished in Section IV.C.

B. Uniqueness Prediction

In this sub-section, C_1 performance is explored, and FUB library creation by the two flows are compared.

The last column in Table I lists the number of possible synthesis configurations of a FUB function according to Eq. 1. With the conventional flow, it is extremely time consuming to extensively explore the configuration space. Thus, six weeks is used as a constraint on the amount of time for synthesis for the conventional flow. In addition, we limit a synthesis configuration to at most eight logic functions.

For the ML-aided flow, the input data samples $\mathbf{X}$ for classifier C_1 are also used by the conventional flow. Instead of directly running synthesis on each configuration, C_1 predicts which configurations will lead to unique implementations. Following the on-line learning described in Algorithm 1, 20,000 synthesis configurations are randomly selected and labeled after synthesis, which forms the initial training data $\mathbf{D}_{\text{train}}$. For each iteration of the while loop, 10,000 top-ranked configurations $\mathbf{X}_{\text{top}}$ are synthesized to obtain their true labels $\mathbf{Y}_{\text{top}}$. Also, the samples with probabilistic labels that are less than 0.01 are added to the bottom-ranked data $\mathbf{X}_{\text{btm}}$.

Table II lists the quantitative results of C_1 and comparison of the two flows. The third and fourth columns show the number of synthesis runs of the two flows while the fifth and sixth columns show the number of unique implementations generated by each flow. In the conventional flow, each configuration is used for synthesis. This means that the conventional flow generates the true labels for all the input configurations of C_1, which provides the ground truth to evaluate C_1's performance. Precision and recall for C_1 are listed in seventh and eighth columns. The precision and recall of the label-1 class (the configuration that leads to a unique FUB implementation) are calculated according to Eq. 2, where precision is calculated as the ratio of column 5 to column 4, and recall is calculated as the ratio of column 6 to column 5.

The recall of classifier C_1 is almost perfect, which means the ML-aided flow identifies virtually all of the unique FUB implementations produced by the conventional flow. The low precision does not impact the efficiency of the ML-aided flow as compared to the conventional flow. Moveover, precision is expected to be low given the extraordinary amount of imbalance in the data. For example, for the Lib0-Cygnus case, row 1 of Table II, shows that only 99,031 out of 7,470,000 samples have label "1", which means that in the conventional flow, for each synthesis run that leads to a unique implementation, **74** others synthesis runs do not. for such an imbalanced case, it is extremely difficult to simultaneously achieve high precision and recall for the minority class. For the ML-aided flow, a 19.8% precision (row 1 of Table II means, on average, only five synthesis runs are needed to produce a unique FUB implementation. This siginifcant change in the amount of synthesis is the source of the speed-up of the ML-aided flow. It is also possible to trade-off precision and recall by the strategies mentioned in [10], but in this application,

TABLE III
PREDICTION OF TESTABILITY FOR VARIOUS UNIQUE FUB IMPLEMENTATIONS.

Library	FUB function	No. of testability analysis		No. of impl. with high testability		Classifier C_2		Analysis reduction
		Conventional flow	ML-aided flow	Conventional flow	ML-aided flow	Precision	Recall	
Lib0	Cygnus	99,031	72,336	26,137	25,940	35.9%	99.3%	1.37×
	Hercules	313,262	225,962	138,644	138,577	61.3%	99.9%	1.39×
Lib1	Cygnus	125,779	84,288	31,092	30,719	36.5%	98.9%	1.50×
	Hercules	406,849	285,109	172,604	172,485	60.5%	99.9%	1.42×

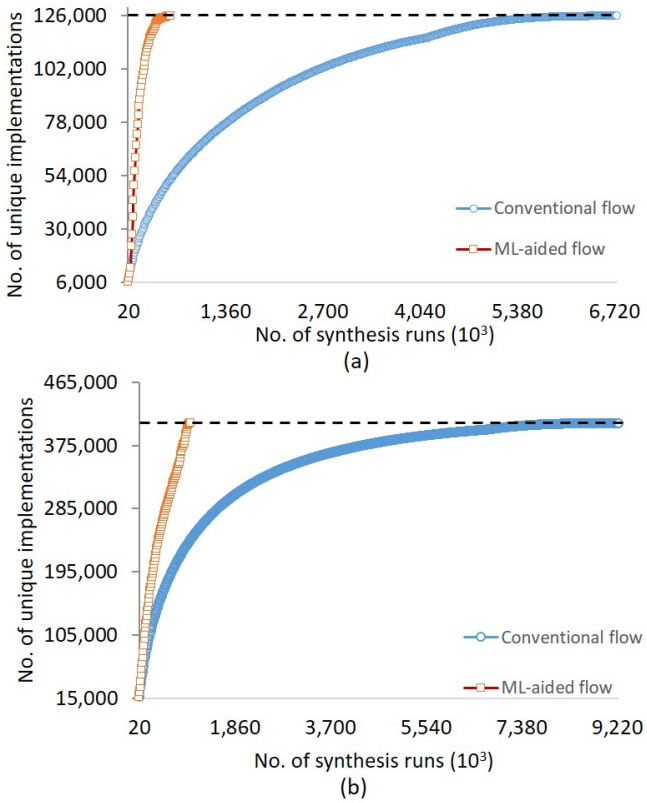

Figure 11. Number of unique implementations generated using standard-cell library Lib1 for FUB functions (a) Cygnus and (b) Hercules.

recall is much more important than precision because it is highly desirable to find every unique FUB implementation. Therefore, we tune the learning so as to achieve a near-perfect recall while still ensuring a satisfactory precision for ensuring efficiency.

With the high performance of C_1 and the on-line learning, significant speed-up for FUB library creation is achieved. The last column in Table II shows the synthesis reduction calculated as the ratio of column 3 to column 4. Speedup is also demonstrated in Figs. **10** and **11**, which show the number of unique FUB implementations as a function of the number of synthesis runs for libraries Lib0 and Lib1, respectively. The ML-aided curves (squares) in Figs. **10** and **11** have a much steeper slope than the conventional curves (circles), and thus reach the same number of unique implementation using significantly less synthesis resources.

C. Testability Prediction

In this sub-section, C_2 performance is explored, and testability analysis of the FUB implementations by the two flows are compared.

Once the FUB library is created, the conventional flow analyzes the testability of every implementation. For the ML-aided flow, the FUB implementations are fed as input to classifier C_2. Instead of directly running ATPG on each FUB implementation as is done in the conventional flow, C_2 predicts which implementations are likely to achieve high testability and then only those implementations are submitted for ATPG. The on-line learning described by Algorithm 1 is also deployed here with different hyper-parameters. To be specific, 10% of the data samples are used as the initial training data for C_2; in each iteration, 5,000 top-ranked implementations (i.e., $\mathbf{D}_{\text{train}}$) are analyzed by running ATPG to obtain their true labels; 0.1 is set as the threshold for collecting the bottom-ranked data samples (i.e., $\mathbf{X}_{\text{btm}}$).

Table III lists the quantitative results of C_2, and comparison of the two flows. The third and fourth columns show the number of ATPG runs for the two flows while the fifth and sixth columns show the number of FUB implementations with high testability. The conventional flow generates the true labels for all data samples of C_2, which provides the ground truth to evaluate C_2's performance. Precision and recall for C_2 are listed in the seventh and eighth columns, respectively. The precision and recall of the label-1 class (the implementation with high testability) are calculated according to Eq. 2. Like C_1, C_2 also achieves near-perfect recall, which means the ML-aided flow identifies almost all FUB implementations with high testability from the FUB library. Similar to C_1, we tune C_2 to achieve high recall rather than precision in order to identify nearly all FUBs with high testability. The last column in Table II shows the reduction in ATPG calculated as the ratio of column 3 to column 4. The reduction in ATPG is not as significant as it was for synthesis. However, it is still very meaningful to deploy C_2 for reducing the resources required for test chip design, especially given the small amount of resources needed to train and utilize classifier C_1 and C_2 (e.g., training C_2 takes less than a second). In other words, the amount of ATPG is reduced 27% ~ 33% with negligible cost.

D. Test Chip Design Evaluation

The last two subsections demonstrate how machine learning can be used to significantly reduce the amount of synthesis and ATPG within the CM-LCV design flow. Here, we complete

TABLE IV
DESIGN EFFORT AND CHARACTERISTICS COMPARISON OF TEST CHIP DESIGNS CREATED BY THE CONVENTIONAL AND ML-AIDED FLOWS.

Library	Cell distribution	Mismatch rate of cell-usage		SSL fault coverage		IP fault coverage		Runtime (hours)	
		Conv. flow	ML-aided flow	Conv. flow	ML-aided flow	Conv. flow	ML-aided flow	Conv. flow	ML-aided flow
Lib0	BlockA	4.9%	4.9%	99.7%	99.7%	79.2%	79.2%	2042.7	188.8
	BlockB	8.7%	8.8%	99.6%	99.5%	75.8%	75.6%	2040.2	188.5
Lib1	BlockC	7.5%	7.5%	99.6%	99.6%	76.8%	76.8%	2054.6	244.7
	BlockD	12.8%	12.8%	99.6%	99.6%	75.9%	75.9%	2064.5	253.8
	BlockE	11.7%	11.8%	99.4%	99.3%	76.1%	76.1%	2061.7	251.0

the remaining steps in the two design flows to obtain the final test-chip designs for comparison.

For both flows, after synthesis and testability analysis, the unique FUB implementations with high testability for the two FUB functions are combined together to form a FUB library. Then, for each cell-usage distribution corresponding to the industrial blocks (BlockA - BlockE), an optimization problem is formulated to select FUB implementations to embody the test chip, with the aim of minimizing the mismatch rate in cell usage while achieving high testability. As a result, five test chip designs are generated using Lib0 and Lib1 to match five objective cell-usage distributions.

Table IV lists the quantitative results of design effort and the characteristics of the LCVs created by the two flows. The third and fourth columns report the amount of mismatch between the LCV designs and the industrial distributions for the two flows. The testability of the designs is evaluated using single stuck-at line (SSL) fault coverage and IP fault coverage [9], which are shown in columns 5-8. Comparing the characteristics of the designs reveals that three out of five created by the ML-aided flow do not suffer from any performance degradation; the remaining two have at most 0.1% degradation in cell mismatch, 0.2% IP fault coverage reduction. These results demonstrate that only a few unique, high-testable FUB implementations are missed by the ML-aided design flow. Finally, the effort required by both flows are measured in terms of compute time. Specifically, the last two columns of Table IV reports the CPU runtime for creating the corresponding designs for the two flows. The reported times include the CPU runtime: (i) FUB library creation, (ii) testability analysis, and (iii) solving the optimization for forming the test chip. The results prove that the ML-aided design flow can provide up to 11× speed-up for a test-chip design with negligible performance degradation.

V. SUMMARY

In this work, a methodology is developed to improve the efficiency of logic test chip design. We develop two RF classifiers to predict (i) whether a synthesis configuration will result in a unique FUB implementation, and (ii) whether a unique FUB implementation has an acceptable level of testability, so that unnecessary synthesis and ATPG can be avoided. In addition, we develop an on-line learning strategy to mitigate the cost of obtaining training data. Various design experiments demonstrate that the ML-aided flow speeds up design by 11× with negligible performance degradation.

REFERENCES

[1] S. Saxena *et al.*, "Impact of Layout at Advanced Technology Nodes on the Performance and Variation of Digital and Analog Figures of Merit," in *2013 IEEE International Electron Devices Meeting*, pp. 17.1–17.4, 2013.

[2] R. C. Leachman and S. Ding, "Excursion Yield Loss and Cycle Time Reduction in Semiconductor Manufacturing," *IEEE Transactions on Automation Science and Engineering*, vol. 8, no. 1, pp. 112–117, 2011.

[3] C. Hess *et al.*, "Stackable Short Flow Characterization Vehicle Test Chip to Reduce Test Chip Designs, Mask Cost and Engineering Wafers," in *IEEE Adanced Semiconductor Manufacturing Conference*, pp. 328–333, July 2017.

[4] M. Bhushan and M. B. Ketchen, *Microelectronic test structures for CMOS technology*. Springer Science & Business Media, 2011.

[5] L. Zhuang *et al.*, "Using Pattern Enumeration to Accelerate Process Development and Ramp Yield," in *Proc. of SPIE*, vol. 9781, 2016.

[6] M. Fujii *et al.*, "A Large-scale, Flip-flop RAM Imitating a Logic LSI for Fast Development of Process Technology," *IEICE transactions on electronics*, vol. 91, no. 8, pp. 1338–1347, 2008.

[7] R. D. Blanton, B. Niewenhuis, and C. Taylor, "Logic Characterization Vehicle Design for Maximal Information Extraction for Yield Learning," in *IEEE International Test Conference*, pp. 1–10, 2014.

[8] R. D. Blanton, B. Niewenhuis, and Z. Liu, "Design Reflection for Optimal Test Chip Implementation," in *IEEE International Test Conference*, pp. 1–10, 2015.

[9] R. D. Blanton and J. P. Hayes, "Properties of the Input Pattern Fault Model," in *IEEE International Conference on Computer Design*, pp. 372–380, 1997.

[10] Q. Huang *et al.*, "Improving Diagnosis Efficiency via Machine Learning," in *IEEE International Test Conference*, pp. 1–10, 2018.

[11] A. B. Kahng, U. Mallappa, and L. Saul, "Using Machine Learning to Predict Path-Based Slack from Graph-Based Timing Analysis," in *IEEE 36th International Conference on Computer Design*, pp. 603–612, 2018.

[12] Z. Liu *et al.*, "Achieving 100% Cell-aware Coverage by Design," in *Design, Automation& Test in Europe*, pp. 109–114, 2016.

[13] Z. Liu *et al.*, "Front-end Layout Reflection for Test Chip Design," in *IEEE International Test Conference*, pp. 1–10, 2017.

[14] Z. Liu and R. D. Blanton, "Back-end Layout Reflection for Test Chip Design," in *IEEE International Conference on Computer Design*, pp. 456–463, 2018.

[15] K. Keutzer, "DAGON: Technology Binding and Local Optimization by DAG Matching," in *24th ACM/IEEE Design Automation Conference*, pp. 617–623, 1987.

[16] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.