Explainable Artificial Intelligence for Better Design of Very Large Scale Integrated Circuits

by

Wei Zeng

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy
(Electrical Engineering)

at the
UNIVERSITY OF WISCONSIN–MADISON
2021

Date of final oral examination: 08/12/2021

The dissertation is approved by the following members of the Final Oral Committee:
Azadeh Davoodi, Professor, Electrical and Computer Engineering
Yu Hen Hu, Professor, Electrical and Computer Engineering
Mikko Lipasti, Professor, Electrical and Computer Engineering
Dan Negrut, Professor, Mechanical Engineering

ABSTRACT

With the advance of Very Large Scale Integration (VLSI) technology, the design process of VLSI circuits becomes more complex, challenging, and time-consuming. Recent years have seen a rising trend of machine learning (ML) incorporated in VLSI design flow for better and more efficient design and implementation of integrated circuits.

Explainable Artificial Intelligence (XAI) is an emerging technique that aims to perform prediction tasks while providing explanations for the predictions. XAI adds transparency and trustworthiness to ML models, leading to better human understanding and exploitation of the models. With ML being applied in VLSI design, it is desirable to adopt ideas from XAI for even better and more trustworthy outcomes of VLSI design.

This dissertation explores the usage of Shapley Additive Explanation (SHAP)—a recent development in XAI, on different aspects and stages of VLSI design flow. Specifically, we propose three techniques that adopt SHAP in front-end and back-end design flows, including (a) SHAP-guided layout obfuscation for enhanced hardware security in split manufacturing, (b) explainable routability prediction, which accelerates the physical design flow and provides hints for improving the design, and (c) explainable-ML-guided approximate logic synthesis for area-efficient computing in error-tolerant applications. These are the first works that incorporate XAI into VLSI design methodology. All of them achieve better results than their conventional counterparts or existing works in similar settings.

ACKNOWLEDGMENTS

First and foremost, I would like to thank Prof. Azadeh Davoodi, whom I am so fortunate to have as my PhD advisor. She has been very helpful and supportive not only academically but also mentally, as well as in my post-graduate career. She is a nice advisor who really cares about her students in all aspects, to an extent far beyond my expectations.

I am very grateful to all my family members, colleagues and friends for their kind supports during my PhD study, especially during the hard times of COVID-19 pandemic. A special gratitude goes to my dearest parents and grandparents for their generous, continuous and unconditional love for decades and counting.

I would like to appreciate Dr. Rasit O. Topaloglu, one of the co-authors of most of my research papers adapted in this dissertation, for his commitment to our regular discussions and constructive suggestions and comments during my course of PhD research.

Also, thanks to all final oral committee members for their valuable feedback on my research proposal and this dissertation.

Last but never the least, I would like to acknowledge Prof. Xuan Zeng, Prof. Xin Li, and Dr. Hengliang Zhu, who had guided me through my very first research in this field even before I started this great PhD journey.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

1 INTRODUCTION AND MOTIVATION

With the advance of Very Large Scale Integration (VLSI) technology, the design of VLSI circuits becomes more complex, challenging, and time-consuming. Recent years have seen a rising trend of machine learning (ML) applied in different aspects of VLSI design and manufacturing, including accelerating the design flows of logic synthesis, optimization and verification [49, 25, 24], physical design and verification [37, 64, 9], as well as in the fabrication stage, such as yield analysis and improvement [31] and lithographic hotspot detection [71], and also in hardware security, including logic locking [7] and split manufacturing [72]. Overall, the incorporation of ML is making the designs more efficient, robust and secure, thus helping reduce the time-to-market, improve reliability, and protect the intellectual property of the products.

Generally speaking, ML performs prediction tasks based on a trained model, which is derived with a set of training data by a specific algorithm. Depending on the problem, different learning paradigms (e.g., supervised learning, reinforcement learning) and types of models (e.g. logistic regression, decision tree, neural network) may be adopted.

On one hand, due to the high complexity of VLSI design tasks, the adopted ML models usually need to be complex enough to achieve good accuracy. On the other hand, in practice, it is always desirable for designers and managers to understand what happens behind the ML model. They may raise questions like, why the model predicts a design failure with a specific parameter set, why the model predicts a lithographic hotspot at a specific location, why the design is predicted as not secure at a specific gate, etc. Therefore, problems of root causing and trustworthy may arise when a ML model is too complex to understand why it predicts a specific data point as such.

In general, the predictive performance of a model is positively corre-

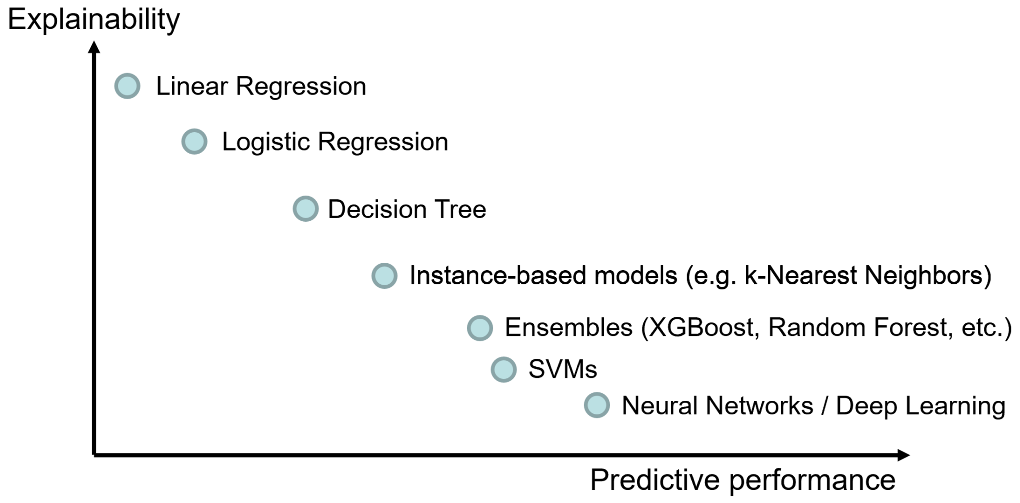

Explainability
Linear Regression
Logistic Regression
Decision Tree
Instance-based models (e.g. k-Nearest Neighbors)
Ensembles (XGBoost, Random Forest, etc.)
SVMs
Neural Networks / Deep Learning
Predictive performance

Figure 1.1: Tradeoff of predictive performance and explainability of different ML models (not to scale).

lated to its complexity, which in turn is negatively correlated to its explainability. This results in a negative correlation of predictive performance vs explainability, as notationally illustrated in Figure 1.1.

Different from pure ML, Explainable Artificial Intelligence (XAI) is an emerging technique that aims to perform predicting tasks and provide *explanations* for the predictions simultaneously. In other words, XAI tries to move the points in Figure 1.1 upwards for a better tradeoff of predictive performance vs explainability.

Besides providing better human understanding and attribution, XAI also enables better exploitation of the model. For example, with the understanding of *why* a design is predicted not secure at a certain location, designers know how to enhance the security in the most effective manner, whereas a pure ML prediction may not provide such insights.

Based on this idea, we propose three novel techniques from different aspects of VLSI computer-aided design (CAD) that utilize XAI, including routing obfuscation for enhanced security for split manufacturing, explainable prediction of design rule check (DRC) violation hotspots in early stages of physical design flow, and a sampling-based approximate logic synthesis technique powered by explainable ML.

In the remainder of this dissertation, we first give an overview of the related works in Chapter 2. Chapter 3 summarizes the contributions of this dissertation. Chapter 4 introduces Shapley Additive Explanations (SHAP)—the technical basis of explanatory analysis adopted throughout this dissertation. In Chapters 5 through 7, we show the three proposed techniques as applications of XAI in VLSI design, focusing on hardware security, layout routability, and approximate logic synthesis, respectively.[1] Finally, in Chapter 8, we conclude this dissertation and discuss future directions.

[1]These chapters are slightly modified versions of works [74, 73, 75] published or to be published in conference proceedings, and have been reproduced here in accordance with the permission of the copyright holders.

2 RELATED WORKS

In this chapter, we first give an overview of related work on applying ML in the VLSI design flow. We note, none of these existing works utilize explainability. Next, we give an overview of existing techniques on explaining ML models in general (and not in the field of VLSI design).

2.1 ML Applications in VLSI Design

ML has been applied in different aspects of VLSI design: to accelerate the design flows, to analyze and improve the robustness in design and manufacturing, and to analyze and enhance the design security against reverse engineering and intellectual property theft.

2.1.1 For Efficient Design Flows

ML plays a role in almost every stage of the VLSI design flow in both front end and back end [28], including logic synthesis, technology mapping, logic optimization, logic verification, placement, and routing. One common usage of ML is to predict the results of a design stage and/or its downstream stages in the design flow, thus reducing the overall runtime of the design flow.

In logic synthesis and optimization, Rokach *et al.* proposed a decision-tree-based circuit decomposition engine for logic synthesis [49]. Pasandi *et al.* proposed a technology mapping approach for approximate logic synthesis based on deep reinforcement learning [41]. Deep reinforcement learning has also been adopted for logic optimization [25, 79]. In logic verification, techniques that accelerate functional coverage closure have been proposed using multiple ML models [24]. Different types of neural networks have been adopted for analyses of testability [35], power [76], signal integrity [15]. In physical design, deep learning and reinforcement learning have been applied to chip placement and legalization [37, 39]. There are works on predicting the routability, in particular, design rule violations, before the detailed routing stage using different ML models [9, 55, 54, 12, 64]. Besides, ML has been used to automate the design space exploration [32] and tuning of flow parameters [63], which helps reduce the design turnaround time.

2.1.2 For Robust Fabrication

For chip fabrication, ML has been applied in VLSI testing [65], yield analysis and optimization of integrated circuits (ICs) [31, 1, 11, 30]. For VLSI in particular, recent focuses are on lithographic modeling and hotspot detection [71, 67, 47], and mask optimization [69, 68].

2.1.3 For Hardware Security

ML has also been applied in the community of hardware security [22], especially in the analysis of spectre attacks [77], IC counterfeit [29], IC reverse engineering [3], logic encryption [7, 53], and split manufacturing [72], etc.

2.2 Studies on Explaining ML Models

There are different methods of explaining the prediction of a ML model to humans [2, 38]. Given a specific prediction/decision from the model, one way to explain it is to analyze important features that contributes most to a specific prediction [48, 34]. Other methods include training an approximated model that has higher explainability [18], learning a separate explanatory model from human-provided explanations [27], etc.

In this proposal, we will focus on the first method, i.e., the attribution of features in predicting a specific data sample. LIME [48] and SHAP [34] are among the most famous explanatory frameworks in this category. They explain individual predictions by evaluating the effects of different features in inferring on each specific data sample. Next, we give a brief overview of these methods.

2.2.1 Local Interpretable Model-Agnostic Explanations (LIME)

LIME [48] is the first work in this category. It finds a local linear approximation of the prediction, such that the most important features for predicting a specific data sample can be identified by observing the coefficients of the linear approximation. In other words, it estimates how much the prediction will change if a feature value changes in its vicinity.

However, LIME is based on local linearity of the ML model, and assumes no interactions among different features. Therefore, it may be inaccurate where the model is complex and/or some features are correlated, as is often the case in practice. To address these, Lundberg *et al.* proposed another additive explanation method named SHAP [34], which does not rely on these assumptions.

2.2.2 Shapley Additive Explanations (SHAP)

SHAP [34] is based on the concept of Shapley value in game theory. It seeks a linear decomposition of contributions made by each feature in predicting a specific data sample. By exploiting the definition of Shapley value, SHAP is reported to be more accurate and more consistent with human intuition, though computationally expensive to calculate exactly. To alleviate the runtime while preserving the accuracy, SHAP tree explainer [33] is later proposed as an extension specifically for tree-based ML models.

This dissertation will be utilizing SHAP analysis in later chapters. More details on SHAP analysis will be provided in Chapter 4.

3 SUMMARY OF CONTRIBUTIONS

In this dissertation, we propose three techniques where XAI is applied in different aspects of VLSI CAD, including logic synthesis, physical design, and hardware security for higher efficiency and effectiveness. To date, the idea of incorporating XAI into VLSI CAD is novel and of great interest to both academia and industry. Specifically, our contributions are summarized as follows.

Routing Obfuscation Guided by Explanatory Analysis of a ML Attack Split manufacturing is a technique to protect the intellectual property related to a design from being stolen by an untrusted foundry, where only a partial layout (e.g., patterns in lower layers) is revealed to an untrusted foundry. Given a lower part of the layout (up to a predefined split level), the ML attack model in [72] tries to find whether a pair of broken vias at the split level are connected in higher metal and via layers that are not available to the attacker.

To thwart this attack, we propose a routing obfuscator that incorporates explanatory analysis on a recent attack powered by a tree-based ML model [72]. The proposed routing obfuscator is to thwart this kind of attack. While the ML attack model reveals that routing features like Manhattan distance of two broken vias are the most important features overall in the attack, it may not be the case for the entire layout of the design. With XAI on the attack model, we can obfuscate the layout and thwart the attack in a more precise and effective manner — we can not only identify and focus on the most vulnerable pairs of vias, but also obfuscate them by just the necessary amount, thereby reducing the overhead while having better performance of obfuscation. Specifically, we propose to adopt SHAP tree explainer [33] to evaluate the feature importance of each pair of vias at or one level below the split level. This explanatory result can guide us not only to identify the most vulnerable ones to the attack, but also to derive the most "efficient" amount to alternate the location or layer of the vias for obfuscation purposes.

Explainable Prediction of DRC Violation Hotspots In VLSI physical design, detail routing is the most time-consuming step, which could take up to days for each run. Several works address this issue by predicting DRC violation hotspots in early design stages. However, they mainly focused on the predictive performance, without providing the reason why it is predicted as a DRC violation hotspot and how to possibly fix it. In fact, from the perspective of designers and project managers in practice, the reason behind a potential violation hotspot is more important than the prediction itself, which pure ML cannot provide.

As one of the applications of XAI, we propose to combine random forest and SHAP tree explainer to build an explainable predictor of DRC violation hotspots in VLSI physical design, based on information from early design stages (e.g. placement and global routing). With XAI powered by SHAP tree explainer, we can figure out the most important features, which can be interpreted as specific key factors for each predicted DRC violation hotspots, such as the overflow of a certain edge in the congestion map, high pin density at a specific location, etc. Designers can use this information to make local adjustments in placement and routing that help alleviate the violations if they are fixable. Since all of these happen before the most time-consuming step of detailed routing, both the length and numbers of design iterations could be reduced, which potentially translates to much faster time-to-market.

Approximate Logic Synthesis Guided by Explainable ML Approximate logic synthesis (ALS) is the process of generating a Boolean circuit that approximates the functionality of an original circuit within a tolerance of error, in trade of better quality of results (usually a smaller area, power and/or delay). Many ALS techniques have been proposed in the past decades [51].

A specific subset of ALS techniques (e.g., [40]) construct an approximate circuit only based on samples of input-output pairs (i.e., entries in a truth table), which are generally achieved by learning a function that generalizes these samples. We refer to this subset of techniques as *sampling-based ALS* in this dissertation. Sampling-based ALS is drawing increasing research interest, as many promising studies and interesting contests are conducted in recent years [10, 13, 4, 52, 45].

As one of the applications of XAI in this field, we proposed a novel sampling-based ALS framework with focus on utilizing explainable ML. We formulate the approximation of a Boolean function as a supervised ML problem, and propose to use explainable ML to guide the model training. Specifically, we measure the importance of each individual primary input bit with respect to the function output based on the SHAP values, which we further utilize to achieve an efficient implementation of Boolean function approximation with minimal loss in accuracy.

4 PRELIMINARIES

In this dissertation, we adopt SHAP [34], a recently proposed explanatory model that explains predictions from a ML model. This chapter introduces the technical basis of SHAP.

4.1 Shapley Value and SHAP

SHAP [34] is a recent advance in ML community. It is based on Shapley value—a concept from game theory.

Shapley value was originally proposed to quantify the contribution of a player in a cooperative game of n players. Let $v(S)$ be the value generated by the cooperation of players in set $S \subseteq N = \{1, 2, \ldots, n\}$ with $v(\varnothing) = 0$, then the Shapley value (contribution) of player j is defined as

$$s(v, j) = \sum_{S \subseteq N \setminus \{j\}} \frac{|S|!(n - |S| - 1)!}{n!} \cdot [v(S \cup \{j\}) - v(S)]. \tag{4.1}$$

Shapley value has many desirable properties. One of them is that the sum of all Shapley values of individual players equals the value generated by the cooperation of all players, formally

$$\sum_{j=1}^{n} s(v, j) = v(N). \tag{4.2}$$

Based on this property, SHAP was proposed as an additive explanation of a ML prediction. Let $f(\mathbf{x}^*)$ be the ML prediction output for a data instance $\mathbf{x}^* \in \mathbb{R}^n$. In SHAP, the cooperation value $v(S)$ is defined as the expected change in prediction output conditioned on feature values indexed by S.

$$v(S) \equiv \mathbb{E}[f(\mathbf{x}) \mid x_S^*] - \mathbb{E}[f(\mathbf{x})], \tag{4.3}$$

where we denote $\mathbb{E}[f(\mathbf{x}) \mid x_S^*] = \mathbb{E}[f(\mathbf{x}) \mid x_j^*, \forall j \in S]$ for simplicity.

The *SHAP value* of a feature j in predicting $\mathbf{x}^*$ is defined by substituting (4.3) in (4.1),

$$c(\mathbf{x}^*, j) = \sum_{S \subseteq N \setminus \{j\}} \frac{|S|!(n - |S| - 1)!}{n!} \cdot \left\{ \mathbb{E}[f(\mathbf{x}) \mid x_{S \cup \{j\}}^*] - \mathbb{E}[f(\mathbf{x}) \mid x_S^*] \right\}. \tag{4.4}$$

Substituting (4.3) in (4.2), we have

$$f(\mathbf{x}^*) = \mathbb{E}[f(\mathbf{x})] + \sum_{j=1}^{n} c(\mathbf{x}^*, j). \tag{4.5}$$

The above equation reveals the essential idea of SHAP—decomposing the prediction output $f(\mathbf{x}^*)$ as the sum of a base value and contributions from each individual feature. Specifically, the base value $\mathbb{E}[f(\mathbf{x})]$ is the expected prediction based on all training data, and $c(\mathbf{x}^*, j)$ is the contribution of feature j of instance $\mathbf{x}^*$. Each contribution $c(\mathbf{x}^*, j)$ can be positive, negative, or zero, which indicates how (and how much) feature j makes the prediction of this specific instance deviate from the average.

4.2 SHAP Tree Explainer for Tree-Based ML Models

Derived from Shapley values, SHAP values in (4.4) have unique properties that lead to consistent explanations [34]. However, exact evaluation of SHAP values is computationally expensive in general due to the exponential time complexity. Therefore, estimations are generally needed in practical use. A recent extension [33], referred to as *SHAP tree explainer*, shows that the *exact* evaluation of SHAP values can be done in polynomial time exclusively for tree-based models (e.g., decision tree, random forest). The SHAP tree explainer does not assume feature independence, as feature

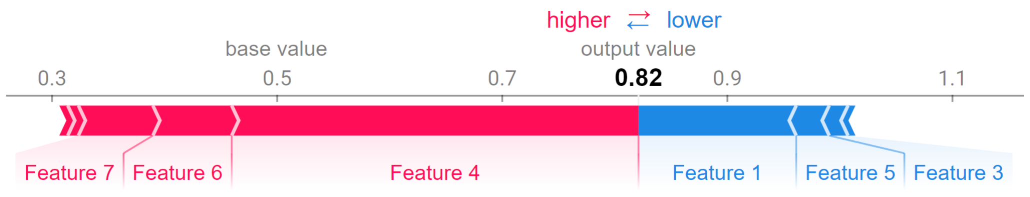


Figure 4.1: An example of force plot from SHAP analysis. Six out of nine features (that contribute the most to the final prediction) are labeled.

interactions are already captured in the underlying trees.

4.3 Examples of SHAP Analysis

In this section, we use an example to show how SHAP analysis works. It can work in different modes. We demonstrate two usages that will be applied in this dissertation.

4.3.1 Explaining a Single Prediction

Suppose in a classification problem, each data instance includes 9 features $x_1, \ldots, x_9$ and the output class $y \in \{0, 1\}$. In the training data set, a half of the instances are of class $y = 1$. A ML model is trained with the training data set and is being used to predict a specific data instance. In this situation, we can gather the base value (the average prediction) and the SHAP values of each feature as defined in (4.4), and visualize them as a *force plot* in Figure 4.1.

In this example, the base value is 0.5 (the same value throughout all data instances), and the prediction of this specific data instance is 0.82. The pink (resp. blue) bars correspond to features that positively (resp. negatively) contribute to this prediction. The lengths of the bars indicate the absolute values of such contributions (i.e. SHAP values). For example, Feature 4 is the most positively contributing feature, indicated by the longest pink bar; Feature 1 is the most negatively contributing feature, indicated by the longest blue bar. Three features that have the smallest contributions (i.e., with the shortest bar lengths on both sides) are not labeled due to space limit. Per (4.5), all bar lengths of positive contributions minus those of negative contributions should equal $d = 0.82 - 0.5 = 0.32$, i.e., the difference of this prediction and the average prediction. The name "force" comes after the metaphor that each feature "pushes" the final prediction from the base value, in the direction indicated by the sign of its SHAP value, by the amount of the magnitude of its SHAP value. The total effect of these "forces" is to push its prediction result from the (common) base value 0.5 to the (instance-specific) output value 0.82.

4.3.2 Comparing Feature Importance

Besides individual explanation, when SHAP is applied to a batch of M instances, we can estimate the *SHAP importance* of each feature. This is done by examining the mean absolute value of SHAP values of a feature implied by the instances. Specifically, the SHAP importance of feature j is defined as

$$Q(j) = \mathbb{E}_{\mathbf{x}} \left| c(\mathbf{x}, j) \right| \approx \frac{1}{M} \sum_{i=1}^{M} \left| c(\mathbf{x}^{(i)}, j) \right|, \tag{4.6}$$

where $\mathbf{x}^{(i)}$ is the i-th instance. Studies have shown that conventional (e.g., impurity-based) feature importance metrics can be biased and not reliable [50], and that SHAP importance is more consistent [33] owing to the properties of Shapley values it bases on.

5 ROUTING OBFUSCATION GUIDED BY EXPLANATORY ANALYSIS OF A ML ATTACK

While recent advances in VLSI manufacturing technology keep pushing the performance boundaries of integrated circuits, the cost of fabricating high-end chips also surges. Consequently, manufacturing outsourcing of VLSI has become more common than ever before. As a result, security issues including design piracy and hardware Trojans injection may arise when an untrusted foundry is involved in manufacturing. To alleviate these problems, split manufacturing is proposed as a technique where the untrusted foundry only receives and fabricates a partial layout up to a metal layer (called a "split layer"). However, this may still not prevent an attacker to extract the full design, if the layout is not obfuscated or if the split layer is too high, as suggested by [46, 59, 36, 72, 58, 66].

5.1 Related Works on Layout Obfuscation

Existing techniques on layout obfuscation may be classified as two categories: placement-based and routing-based. Placement-based techniques include pin swapping [46], cell insertion [62], and cell location perturbation [59]. Routing-based techniques include routing blockage insertion [36], routing perturbation [60], and wire lifting [43]. The two techniques may also been combined, as in [42].

The key idea of design obfuscation for split manufacturing is to make an attack model fail to identify correct connections above the split layer. As for the attack models for split manufacturing, Rajendran *et al.* first proposed the proximity attack [46]. Wang *et al.* proposed a more advanced network-flow-based proximity attack [59], which employs the network flow model that considers more heuristics for better attack performance. Magaña *et al.* proposed a congestion based attack [36], which redefined proximity measures based on the observation that placement and routing congestions are better indicators in large commercial designs. Recently, a ML-based attack model [72] is proposed, which is trained with empirically-selected layout features that reflect the hints from routing conventions.

In this chapter, we present a novel way to build an obfuscator for split manufacturing based on XAI. We adopt SHAP to analyze the ML attack model in [72]. This ML attack model is especially suitable for large commercial designs while other attack models (e.g. [59]) would take prohibitively long attack time.

The SHAP-based analysis reveals to what extent *each* layout feature contributes to correctly predicting *each* individual unknown connection as seen by an untrusted foundry. We then exploit this information to design a SHAP-guided obfuscator against the ML attack model where only truly vulnerable connections are identified and each is obfuscated by just the necessary amount. This results in minimal perturbation to the layout as measured by increase in wirelength and number of perturbed nets. Our obfuscator (named ObfusX) is routing-based and is performed by utilizing via perturbation and wire lifting schemes. (Placement-based obfuscation was not found to be as effective by our SHAP-based analysis.)

ObfusX sets an example of how XAI can be used to obfuscate a design; while we focus on routing obfuscation for a ML-based split manufacturing attack, our approach is generalizable to build any obfuscator as long as a ML attack model is available. We demonstrate the benefits of ObfusX in identifying and focusing on the most vulnerable candidates and obfuscating each by just the right amount, thereby reducing the obfuscation overhead, while having better performance. Our results are compared with two prominent prior works, using not only the ML attack [72], but also an independent network flow-based attack from a recent work [59].

5.2 Overview of SHAP-Guided Routing Obfuscation

The core idea of a SHAP-guided obfuscation is to perturb the design, such that a ML attack model would perform worse. As we will show in experiments, such obfuscation also performs well under an independent, non-ML attack model [59]. A flow chart of the overall process of ObfusX is shown in Figure 5.1.

The upper panel shows how the ML model is developed. To generate the training set and testing set for a design to obfuscate (i.e., "target design"), we generate data samples by extracting layout features from routed designs, with the same split layer applied as will be used in manufacturing. All data samples from the target design are allocated in the testing set, which we will use to monitor the progress and performance of obfuscation. Other designs in the same benchmark suite as the target design are used to generate the training set that will be used to train the attack model. ObfusX uses the ML predictor in [72], whose details will be summarized later in Section 5.3. With a trained attack model, it predicts how likely each pair of (two) v-pins in the target design could be a match (i.e., are actually connected), which can be interpreted as the vulnerability of the pair to the ML attack.

To develop ObfusX, as shown in the lower panel, the ML prediction for a v-pin pair is fed to the SHAP tree explainer, which generates a set of SHAP values to explain the prediction.

Each SHAP value corresponds to an extracted feature and quantifies to what extent that feature contributes to the ML predictor for that specific v-pin pair. These SHAP values are next analyzed across all actually-connected v-pin pairs to identify the most vulnerable ones to the ML attack, along with the layout features that contribute the most to their individual vulnerabilities.

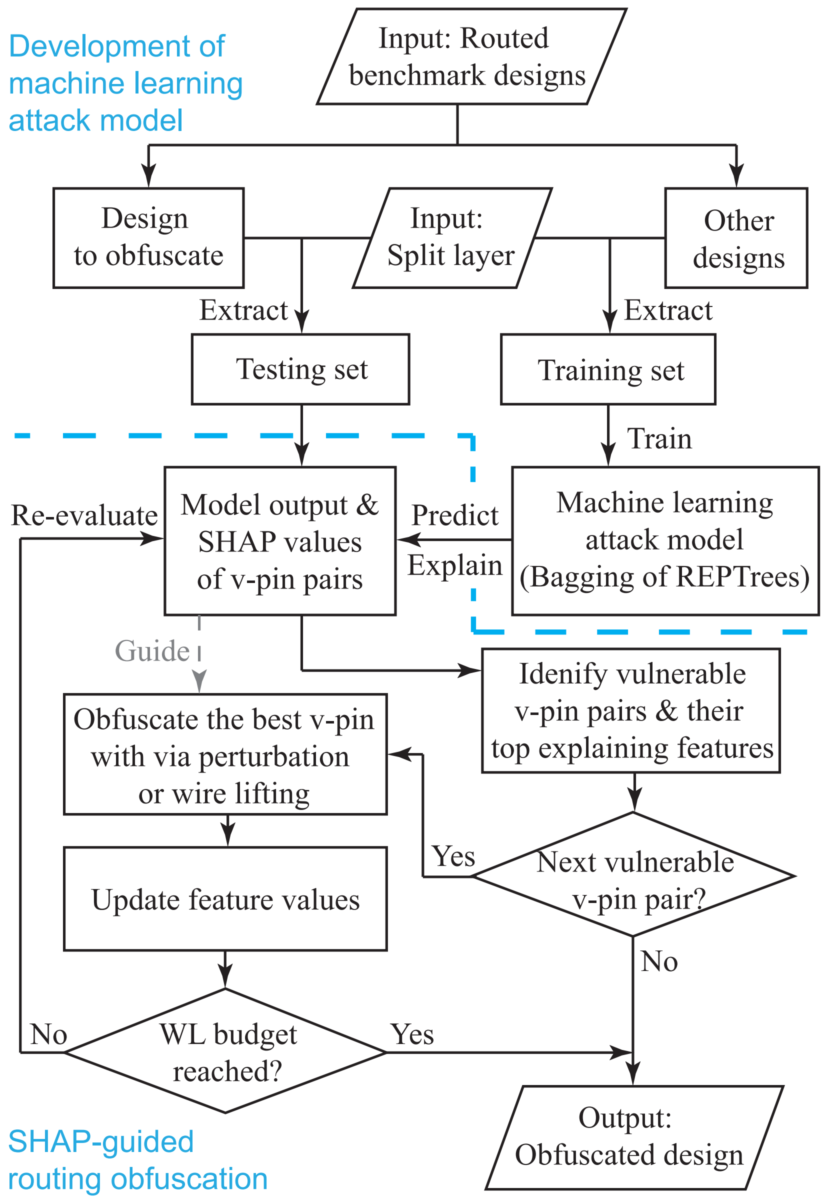

Development of machine learning attack model
Input: Routed benchmark designs
Design to obfuscate
Input: Split layer
Other designs
Extract
Extract
Testing set
Training set
Train
Re-evaluate
Model output & SHAP values of v-pin pairs
Predict
Explain
Machine learning attack model (Bagging of REPTrees)
Guide
Idenify vulnerable v-pin pairs & their top explaining features
Obfuscate the best v-pin with via perturbation or wire lifting
Update feature values
Yes
Next vulnerable v-pin pair?
No
No
WL budget reached?
Yes
SHAP-guided routing obfuscation
Output: Obfuscated design

Figure 5.1: Flow chart of ObfusX.

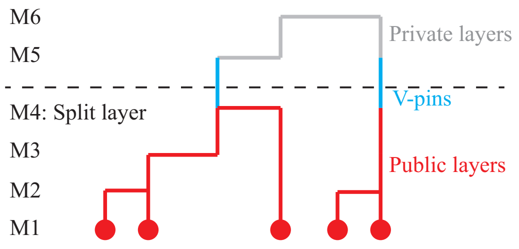


Figure 5.2: Illustration of public/private layers and v-pins in split manufacturing. The split layer is M4.

Next, the output of SHAP analysis guides the actual obfuscation which is done iteratively. ObfusX utilizes two layout perturbation techniques–via perturbation and wire lifting–each of which effectively change the routing and locations of a vulnerable v-pin pair. At each iteration, the most vulnerable v-pin pair is obfuscated if its obfuscation does not violate routing feasibility. Next, the feature vector of the obfuscated pair is updated and consequently its vulnerability is re-evaluated by the attack model (given that the layout has been slightly perturbed). ObfusX then proceeds to obfuscate the next vulnerable pair, until there is no more vulnerable pair, or a budget of wirelength (WL) overhead is reached.

5.3 ML Attack Model for Split Manufacturing

To build an obfuscator, we use SHAP explanatory analysis to break a ML-based attack. Here, we review the ML attack model used by our work.

Given a metal layer as the split layer, the layout is partitioned into public layers, v-pins (as termed in [72]) and private layers from low to high levels. Specifically, as illustrated in Figure 5.2, a *split layer* refers to the topmost metal layer available to the attacker; *public layers* refer to all metal on or below the split layer and via layers in between; *private layers* are all metal layers above the split layer and the via layers in between; *v-pins* are vias connecting public and private layers. The attacker has access to the layout (cells, pins, wires, vias) in public layers and all v-pins. The goal of the split manufacturing attack is to predict the connectivity on private layers based on the available layout on public layers.

Recently, a ML-based attack model was proposed for split manufacturing in [72]. To build the ML model, for each pair of v-pins in a design, first a vector of layout "features" was extracted from the public layers. Using these features, the ML model was built based on Bagging of 10 reduced error pruning trees (REPTrees) in Weka [23]. The ML model mapped each v-pin pair with feature vector x to a probability $f(\mathbf{x}) \in [0, 1]$, indicating how likely the v-pin pair is a "match" (i.e. actually connected to each other on private layers).

For each v-pin v, we record the the v-pin's coordinates on the split layer, denoted by (vx, vy). We compute wirelength W for the route fragment that connects v to one or more pins of standard cells on the underneath placement layer. We also calculate the location where the v-pin connects on the placement layer, which we denote by (px, py). If the connection is to multiple pins on the placement layer, the location is computed by averaging the coordinates of pins of the standard cells that connect to v. We also record the areas of its driving cells and its loads in the placement layer that connects to the v-pin, denoted $OutArea$ and $InArea$, respectively.

Then for a pair of v-pins, we consider the following features extracted in [72]. These features are used to train the ML attack model.

- `DiffPinX` = $|px_1 - px_2|$: This feature records the difference in the x-coordinates of the pins on the placement layer which connect to the two v-pins.

- `DiffPinY` = $|py_1 - py_2|$: Same as the previous feature except calculated using the y-coordinates.

- `ManhattanPin` = $|px_1 - px_2| + |py_1 - py_2|$: Same as the previous feature except it is calculated based on the pin locations on the placement layer.

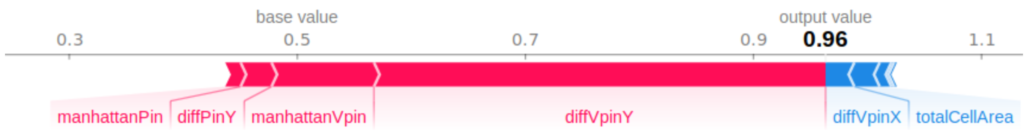
- `DiffVpinX` = $|vx_1 - vx_2|$: This feature records the difference in the x-coordinates of the two v-pins.
- `DiffVpinY` = $|vy_1 - vy_2|$: Same as the previous feature except calculated using the y-coordinates.
- `ManhattanVpin` = $|vx_1 - vx_2| + |vy_1 - vy_2|$: This feature records the Manhattan distance between two v-pins.
- `TotalWirelength` = $W_1 + W_2$: This is the known wirelength connecting the v-pin pair below the split layer.
- `TotalArea` = $InArea_1 + InArea_2 + OutArea_1 + OutArea_2$: This records the sum of cell areas connecting to the two v-pins.
- `DiffArea` = $(OutArea_1 + OutArea_2) - (InArea_1 + InArea_2)$: This feature calculates the area difference of the driving cells from its loads.

In this application, we will use SHAP tree explainer on this ML model to analyze the vulnerability of individual v-pin pairs to the attack, and use it to guide the obfuscation.

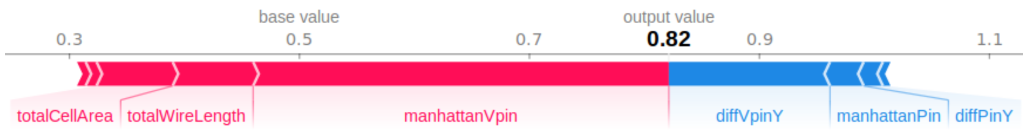
5.4 SHAP Analysis for One V-pin Pair

Before discussing the details of ObfusX, we first explain how SHAP-based analysis is performed for a single pair of connected v-pins. This helps us to illustrate the true benefits of such analysis in building ObfusX.

Consider two connected v-pins from the design `superblue1` with split layer M6. The ML attack model, predicts the first pair to be connected with probability 0.96 (which is a relatively high prediction indicating a successful attack if there is no obfuscation). Figure 5.3(a) shows the *force plot*



(a)



(b)

Figure 5.3: SHAP force plots of two actually-connected v-pin pairs. The pink/blue bars quantify to what extent each layout feature positively/negatively contributes to the ML attack that predicts their connectivity. The top contributing features (longest pink bars) may vary from one v-pin pair to another. For example, `diffVpinY` is the most contributing feature in predicting (a) (longest pink bar) while it is actually the most negatively contributing feature to predicting (b) (longest blue bar).

generated by SHAP analysis performed on the ML prediction for this pair. The color and length of pink/blue bars show the signs and magnitudes of each contribution $c(\mathbf{x}, j)$ in Equation (4.5), respectively. For the pair in Figure 5.3(a), the analysis breaks down the prediction output of 0.96 as sum of a base value of 0.5 and a total deviation of $+0.46$. The pink/blue bars correspond to the features which positively/negatively contribute to the model output (i.e., with a positive/negative "force" pushing towards this 0.96 prediction). The length of the bars indicate the degree of contribution such that the sum of the lengths of pink bars (with positive sign) and blue bars (with negative sign) adds up to $+0.46$.

More specifically, for pair (a), among all its features, `diffVpinY` has the highest SHAP value of around $+0.4$ (corresponding to the length of its pink bar). Figure 5.3(b) shows the force plot for a second pair (b). For pair (b), we observe a different feature, i.e., `manhattanVpin` is dominant. Moreover, `diffVpinY`, which was the top feature in (a), has a negative

SHAP value in (b), indicating it actually contributes negatively to the prediction of pair (b).

In fact, for pair (a), if we were to increase the values of `diffVpinY` and `manhattanVpin` by the size of one routing grid (which simulates a via perturbation in y direction), the output (i.e., the probability of matching) from the ML attack model would drop significantly from 0.96 to 0.37. However, if the same were done for pair (b), the model output would only have a small drop from 0.82 to 0.64. It confirms that `diffVpinY` is a useful feature from an obfuscation perspective for pair (a), but is less useful for pair (b).

The above example yields the following two key observations to illustrate the unique benefits of SHAP analysis for obfuscation:

1. The vulnerable v-pin pairs can be identified as the ones which have few features with large positive SHAP values.
2. The top feature may vary across individual pairs, implying a different degree or scheme of obfuscation is needed for each.

These observations motivate us to develop ObfusX, a SHAP-based obfuscator which decides, precisely, how much each v-pin pair should be uniquely obfuscated for most efficiency.

5.5 Details of SHAP-Guided Routing Obfuscation

The goal of SHAP-guided obfuscation is to alter the SHAP values such that there will not be any dominant feature with a high positive SHAP. It could mean that obfuscation makes originally dominant features to have a lower positive SHAP value or a negative one.

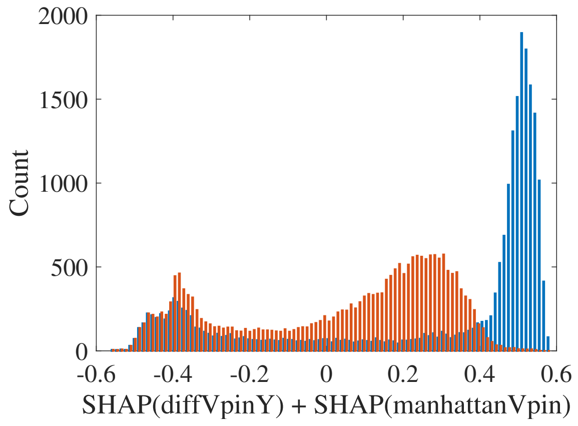


Figure 5.4: Contributions of top two features `diffVpinY` and `manhattanVpin`, shown as a distribution for all connected v-pin pairs, before (blue) and after (red) obfuscation. ObfusX flattens the distribution and decreases the top contributions.

Our SHAP analysis of design `superblue1` with split layer M6 shows that, for about half of the connected v-pin pairs, the SHAP value of `diffVpinY` is consistently dominant (followed by that of `manhattanVpin`). However, for the other half of pairs, the distribution of SHAP values over features becomes fuzzy, which suggests that no single feature dominates the model. Such pairs (which do not have any dominant feature) do not need to be obfuscated.

For the two dominant features (`diffVpinY` and `manhattanVpin`) in the above example, Figure 5.4 shows the distribution of the combined contribution (i.e., sum of SHAP values) of these two top features, before and after obfuscation. This is when using ObfusX with via perturbation (which will be discussed in detail in Section 5.5.1). The before-obfuscation distribution is shown in blue and the after-obfuscation one is shown in red. As can be seen, ObfusX flattens the distribution and shifts it to the left (so it decreases the top contributions, making some less positive and some even negative).

Similar to the example of `superblue1` with split layer M6, SHAP-based analysis with the rest of the designs showed that `diffVpinY` and `manhattanVpin` are always the top two contributing features for many of the vulnerable nets when the split layer is even. (For odd split layers, `diffVpinY` should be replaced by `diffVpinX` because wires are preferred to route vertically on even layers and horizontally on odd layers.) The nets which did not have a dominant feature simply will not need to be obfuscated with SHAP-guided analysis. Therefore, these two features are the only ones utilized by ObfusX.

We note, these two dominant features are related to routing which explains our choice to obfuscate the design with routing-based techniques, i.e., via perturbation and wire lifting. However, we note, our general approach is not restricted to routing.

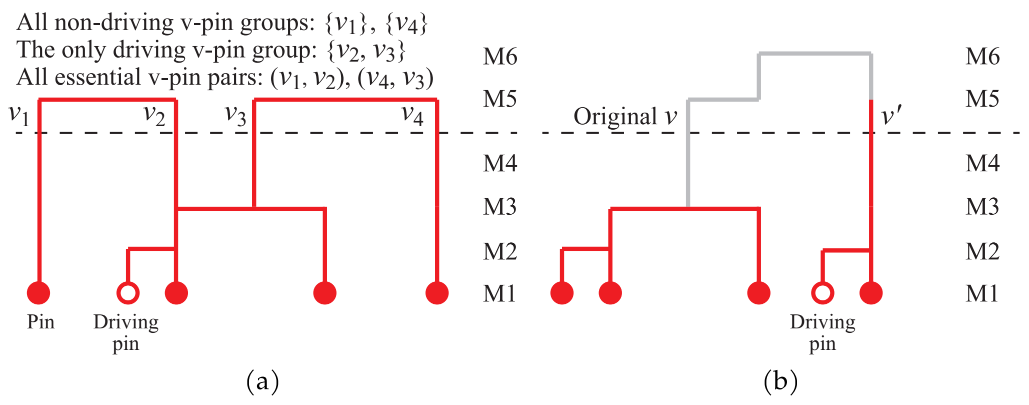
Next, we explain the two routing-based techniques used by ObfusX.

5.5.1 ObfusX with Via Perturbation

The procedure for via perturbation only considers perturbing v-pin pairs which are determined to be "essential". Essential v-pin pairs are a subset of all connected v-pin pairs, after disregarding trivial cases, e.g., when some v-pins connect to each other using the public layer, which are easily identifiable by the attacker. ObfusX also ensures feasibility of the routing throughout the process without any area overhead. We first introduce the following which will be used when explaining the algorithm.

5.5.1.1 Terminology

We introduce the following terminology as shown in Figure 5.5(a), where the split layer is M4, public layers are M1 through M4, and private layers are M5 and M6. Wires in all metal layers are shown as horizontal lines and vias as vertical lines.


All non-driving v-pin groups: $\{v_1\}$, $\{v_4\}$
The only driving v-pin group: $\{v_2, v_3\}$
All essential v-pin pairs: (v_1, v_2), (v_4, v_3)
v_1
v_2
v_3
v_4
M6
M5
M4
M3
M2
M1
Pin
Driving pin
(a)
Original v
v'
Driving pin
(b)

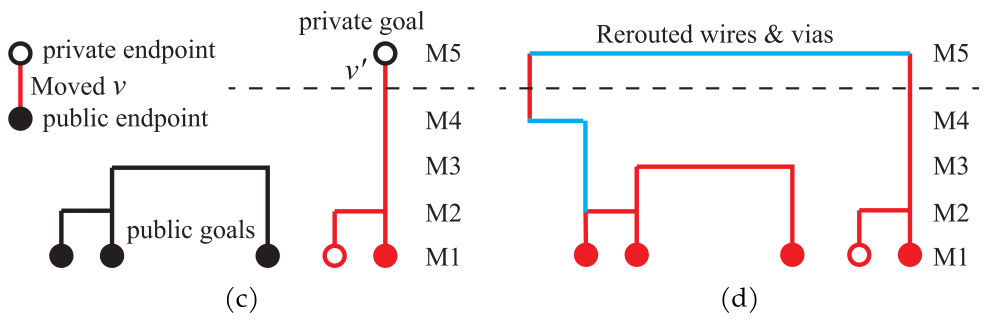


Figure 5.5: (a) Illustration of terminology. (b–d) Rip up and reroute for v-pin pair (v, v') when v is perturbed. The horizontal lines in the illustration correspond to wires (which can be in x- or y-direction) and the vertical lines correspond to vias. The dashed lines separate the EOL and BEOL. (b) Original wires and vias of the net containing v and v'; the gray segments are to be removed. (c) The new location of v after perturbation is identified. The unconnected parts (including both endpoints of v and rerouting goals) are identified in the public layers (shown in black wires and dots) and private layers (shown in black circles). (d) The unconnected parts are reconnected (in blue) using public and private layers respectively, with A* search algorithm in 3D grids.

A *driving pin* is a pin that drives other components in the net. It can be the output pin of a logic cell or that of a primary input. Note that there is exactly one driving pin in each net, unless in rare cases where tri-state logic is involved.

A *v-pin group* consists of v-pins in the same net that connect to each other using public layers. The v-pins in the same group can be easily identified by an attacker because they are connected in public layers that are available to the attacker.

A *driving v-pin group* is a v-pin group that connects to a driving pin using public layers. In general, since each net has exactly one driving pin, there is exactly one driving v-pin group in each net.

A *non-driving v-pin group* is a v-pin group that does not connect to any driving pin in public layers.

An *essential v-pin pair* (v, v') consists of a pair of v-pins, where v is in a non-driving v-pin group G, and v' is in a driving v-pin group G'. If G' has more than one v-pin, v' is the closest v-pin to v in G'.

5.5.1.2 Algorithm

We propose an algorithm that perturbs the locations of v-pins based on SHAP values of the top features `manhattanVpin` and `diffVpin`*R* where R is X for odd split layers and Y for even split layers. This is done iteratively, one v-pin at a time. We first calculate the SHAP values $S(i, j)$ for all essential v-pin pairs i and all features j. Then for each essential v-pin pair i, we take the maximum of the SHAP values over all features j, i.e., $S_{\max}(i) = \max_j S(i, j)$.

For efficiency considerations, we only perturb "eligible" v-pins, which satisfy all of the following criteria:

- The v-pin belongs to an essential v-pin pair $p = (v, v')$, with v and v' in the same net. This is to avoid duplicated or invalid perturbations, e.g. perturbing the same v-pin later when a different v-pin pair is being considered.

- $S_{\max}(p) = S(p,\ \texttt{manhattanVpin})$ or $S_{\max}(p) = S(p,\ \texttt{diffVpin}R)$. This ensures the essential v-pin pair p is vulnerable, i.e., likely predictable with the top features.
- $S(p,\ \texttt{diffVpin}R) \geq S(p,\ \texttt{diffVpin}R')$, where $R' \in \{X, Y\}$ is the routing direction other than R. This condition ensures the effectiveness of perturbing v or v' in R direction.
- If there are more than one non-driving v-pin group in the net of v and v', then v' in the driving v-pin group is not eligible for perturbation and only v may be perturbed. Otherwise, perturbing v' may affect multiple essential v-pin pairs at the same time.

The procedures of SHAP-guided via perturbation are summarized in Algorithm 1. We maintain a list $\mathcal{L}$ of essential v-pin pairs $p = (v, v')$ sorted in decreasing order of $S_{\max}(p)$. As shown in Algorithm 1 (lines 7–8), in each iteration, we select p from the top of the list, and apply trial perturbing moves (a series of "dry runs" that do not actually perturb) to each eligible v-pin in pair p within a predefined small radius r (detailed in lines 23–36) to find the most efficient move (v^*, δ^*) which means to move v-pin v^* by amount δ^*. Efficiency of a move is defined in terms of the decrease in the model output $-\Delta f(\mathbf{x})$ and the extra WL ΔWL (as an integer). Specifically, to quantify the efficiency of a move, we define its *gain* as

$$gain = \begin{cases} -\Delta f(\mathbf{x})/\Delta WL, & \text{if } \Delta f(\mathbf{x}) < 0 \text{ and } \Delta WL \geq 1 \\ 1 - \Delta f(\mathbf{x}), & \text{if } \Delta f(\mathbf{x}) < 0 \text{ and } \Delta WL \leq 0 \\ 0, & \text{if } \Delta f(\mathbf{x}) \geq 0 \text{ or not feasible} \end{cases}, \tag{5.1}$$

which prioritizes moves that lead to a decrease in model output at no or low extra cost of WL.

The trial perturbing is necessary as it would be difficult to estimate the routing feasibility and extra WL without any trials due to complex layout congestion. After the trial perturbing, if there is no feasible move[1], we remove pair p from $\mathcal{L}$ (line 19), and proceed to the next v-pin pair in $\mathcal{L}$; if there is any feasible move (lines 10–18), we take the actual move that has the highest gain, update the feature vector and the SHAP values (as in Figure 5.1), re-check the v-pin eligibility, and go to the next iteration.

5.5.1.3 Rip-up and Reroute Procedure

To apply a perturbing move to a v-pin v, we rip up and reroute the wires connecting v to the other components. To facilitate the rerouting procedure, we rip up v and all wires connecting to v that do not result in more than two connected components, while not touching any other v-pins, as shown in Figure 5.5(b). Then we move v to the new location and identify the unconnected parts (i.e. both endpoints of v and the other connected components of the net, referred to as "rerouting goals") in the public and private portions, respectively, as in Figure 5.5(c). Finally, we use A* search algorithm [26] to reconnect the unconnected parts of the net in the public portion using public layers, and then reconnect for the private portion using private layers, as shown in Figure 5.5(d). Specifically, the routing graph $G(V, E)$ for A* search is built in three dimensions. The vertices are valid routing grids in all metal layers, and the edges are in x, y and z directions, corresponding to potential wires (in x and y directions) and vias (in z direction) where the routing resources permit. The edge set E changes dynamically as the routing resources are occupied or released in rip-up and reroute. This rip-up and reroute procedure ensures a feasible route (if possible) and optimizes the WL. Also, for via perturbation, the number of v-pins is not changed after rip-up and reroute, as we reconnect

[1]We say a move is feasible if it does not violate routing resources, i.e., does not cause any congestion overflow in global routing, or any short in detailed routing.

Algorithm 1 Via Perturbation

```
1: procedure Via-Perturbation($\mathcal{L}$, $R$, $r$, $N$)
2:     Input: $\mathcal{L}$: list of all essential v-pin pairs, $R$: perturbing direction, which is $X$ for
       odd split layer and $Y$ for even split layer, $r$: radius for trial perturbing, $N$: maximum
       number of iterations.
3:     for $iter \leftarrow 1$ to $N$ do
4:         if $\mathcal{L}$ is empty then
5:             break
6:         end if
7:         for $p$ in $\mathcal{L}$ in descending order of $S_{\max}(p)$ do
8:             $(v^*, \delta^*) \leftarrow$ Trial-Perturbing ($p$, $R$, $r$)
9:             if $v^* \neq$ null then                              ▷ take the actual move
10:                Ripup-and-Reroute ($v^*$, $R$, $\delta^*$)
11:                Update the feature vector and SHAP values of $p$
12:                Re-check the eligibility of both v-pins in $p$
13:                if neither v-pin is eligible then
14:                    Remove $p$ from $\mathcal{L}$
15:                end if
16:                Re-sort $\mathcal{L}$ by $S_{\max}$
17:                break                                   ▷ only move one v-pin at a time
18:            end if
19:            Remove $p$ from $\mathcal{L}$
20:        end for
21:    end for
22: end procedure
23: procedure Trial-Perturbing($p$, $R$, $r$)
24:    $v^* \leftarrow$ null, $\delta^* \leftarrow$ null
25:    maxGain $\leftarrow 0$
26:    for eligible $v$ in v-pin pair $p$ do
27:        for $\delta \leftarrow -r$ to $r$ do
28:            $gain \leftarrow$ Ripup-and-Reroute ($v$, $R$, $\delta$)      ▷ move $v$ in $R$-dir by $\delta$
29:            if $gain >$ maxGain then
30:                $v^* \leftarrow v$, $\delta^* \leftarrow \delta$
31:                maxGain $\leftarrow gain$
32:            end if
33:        end for
34:    end for
35:    return $(v^*, \delta^*)$                          ▷ Best v-pin to move & the amount
36: end procedure
37: procedure Ripup-and-Reroute($v$, $R$, $\delta$)
38:    Rip up $v$ and any wire connecting to $v$ that does not result in more than two
       connected components or touch other v-pins. (Figure 5.5(b))
39:    Move $v$ in $R$ direction by amount $\delta$
40:    Identify unconnected parts for rerouting (Figure 5.5(c))
41:    Build/update the routing graph and reroute using A* search algorithm (Fig-
       ure 5.5(d))
42:    Calculate $gain$ according to (5.1)
43:    return $gain$
44: end procedure
```

the net using public and private layers separately.

5.5.2 ObfusX with Wire Lifting

Wire lifting is the second routing-based technique in ObfusX. It moves wires from the public layers to private layers, and therefore creates more v-pins, which can make the attack more difficult.

Here, the same flow in Figure 5.1 is followed. However, instead of going through the v-pins connecting public and private layers as in via perturbation, we now consider the vias *one layer below* (i.e. the vias connecting the topmost public metal layer and the metal layer immediately below it). The goal of wire lifting is to make it most difficult for the attack model to identify the created v-pin pairs as connected, after lifting. To this end, ObfusX iteratively selects the via v on this layer which, when lifted above the split layer, would create an essential v-pin pair p whose maximal SHAP value $S_{\max}(p)$, is the lowest among all options of v.

After we select the v-pin v at each iteration, we perform the wire lifting by applying the same rip up and reroute procedure to v as in Section 5.5.1.3, except that,

(a) we do not move the location of v after ripping up for saving WL, and

(b) when rerouting with A* search, we put a higher weight on wires in public layers, so that the use of public wires is discouraged and thus extra v-pins are created.

5.6 Experimental Results

We obtained the source code of the ML attack from [72], used the `shap` library for Python for SHAP analysis, and implemented all procedures of ObfusX in C++. Experiments were done on a Linux workstation with an Intel 16-core 3.60 GHz CPU and 64 GB memory.

Table 5.1: Results of via perturbation on five ISPD'11 benchmark designs

Design	No obfuscation	[72]					ObfusX				
(#v-pins)	HR: 0.01%/0.1%	HR	ΔWL%	PN%	PV%	t_{CPU}	HR	ΔWL%	PN%	PV%	t_{CPU}
Split layer: M6											
sb1 (44486)	23.79 / 63.33	2.19 / 11.58	3.03	99.83	99.58	3.86	0.52 / 6.12	0.55	66.57	36.01	3.28
sb5 (60034)	29.47 / 63.96	5.75 / 20.38	4.09	96.81	91.75	7.13	4.34 / 15.46	0.67	55.62	30.08	5.30
sb10 (89846)	31.84 / 64.34	10.24 / 28.31	4.52	92.45	79.77	7.75	9.37 / 23.93	0.71	46.49	23.96	8.05
sb12 (80816)	33.01 / 75.58	8.23 / 24.78	3.31	97.70	90.12	6.46	4.32 / 11.67	0.64	73.87	37.12	5.45
sb18 (36026)	20.06 / 66.11	4.27 / 16.55	2.64	98.91	94.35	2.88	2.16 / 8.68	0.67	63.02	34.27	2.06
Average	27.63 / 66.66	6.14 / 20.32	3.52	97.14	91.11	5.62	4.14 / 13.17	0.65	61.11	32.29	4.83
Split layer: M4											
sb1 (150510)	49.82 / 68.33	6.46 / 25.37	9.50	99.79	93.91	9.00	1.70 / 24.08	2.14	65.23	35.26	18.90
sb5 (179844)	38.78 / 60.40	7.54 / 23.84	9.86	96.94	87.87	11.48	3.03 / 23.35	1.87	51.43	28.09	18.41
sb10 (200896)	33.50 / 60.21	13.16 / 37.36	8.53	91.38	73.21	15.05	9.81 / 36.54	1.31	38.81	19.55	17.19
sb12 (173294)	47.07 / 71.52	9.01 / 22.40	7.61	98.61	92.32	13.48	4.42 / 17.39	1.12	65.32	32.81	18.09
sb18 (86658)	29.83 / 59.89	5.15 / 17.89	6.43	99.37	95.29	4.26	1.87 / 10.95	1.53	57.00	30.80	7.18
Average	39.80 / 64.07	8.26 / 25.37	8.39	97.22	88.52	10.65	4.17 / 22.46	1.59	55.56	29.30	15.95

5.6.1 Via Perturbation with ObfusX

We first show in Table 5.1 the performance of via perturbation with ObfusX using five designs in ISPD'11 benchmark suite that are also used in [36, 43, 72]. We obtain routed overflow-free designs from [72], to which we apply the proposed SHAP-based via perturbation, with parameter $r = 3\times$ routing grid size. We compare the performance and the cost of obfuscation with the via perturbation technique proposed in [72]. This is based on the same ML attack model[2].

We use the following metrics to evaluate the performance and the cost of an obfuscation[3].

- Hit rate (HR) at X%: For a v-pin v, we first identify the top X% of other v-pins u which have the highest ML model output for essential v-pin pair (v, u). These v-pins are predicted by ML to most likely

[2]Note that the popular network flow attack model [59] takes prohibitively long time to run on these designs and hence is not applicable here.

[3]Note that the functions of standard cells are not available in ISPD'11 benchmark. Therefore metrics related to circuit outputs (e.g. Hamming distance (HD), output error rate (OER)) are not applicable.

be the match for v. We call it a "hit" of v if its real matching v-pin is among the v-pins identified above. We then report the average percentage of hits of all v-pins v in the design.

For each design, we report two HR values with X $= 0.01$ and 0.1, respectively. (As a point of reference, X $= 0.1$ results in up to 89 v-pins identified on split layer M6, or up to 200 v-pins on split layer M4 in these designs. The total number of v-pins is quite large as reported in the first column of the table.) *A lower HR means better obfuscation.*

- WL overhead (ΔWL%): percentage of increase in WL after the obfuscation. *Lower is better.*
- Perturbed nets (PN%): number of perturbed nets divided by total number of nets that *contain any v-pin*. *Lower is better.*
- Perturbed v-pins (PV%): number of perturbed v-pins divided by the number of v-pins in the design. *Lower is better.*
- Total CPU time (t_{CPU}, in hours) for running ObfusX.

Several observations can be made from the results in Table 5.1. **First**, the HR of the ML model for 0.01% and 0.1% v-pin lists drops drastically after obfuscation; for ObfusX it drops from 28% and 67% to 4% and 13%, respectively, better than the HR reductions with in [72]. **Second**, the WL overhead of ObfusX is less than $1/5$ of that with [72]. **Third**, with ObfusX, only around 30% of v-pins and 60% of nets (that contain v-pins) are finally perturbed, compared to nearly-all nets and v-pins when perturbed with [72].

To observe the tradeoff between performance and cost of obfuscation, we plot in Figure 5.6 the curves of HR and WL overhead with ObfusX and [72], respectively. Compared to [72], ObfusX achieves 87% and 97% lower HR in 0.1% and 0.01% v-pin lists, respectively, for the same WL overhead

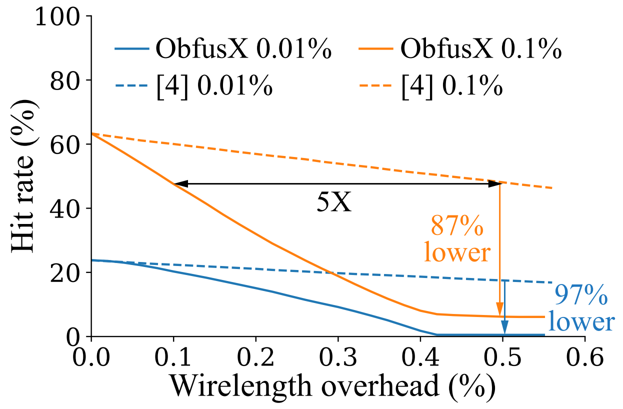


Figure 5.6: Comparison of tradeoff in HR vs WL in `superblue1`.

Table 5.2: Results of wire lifting on six ISCAS′85 benchmark designs

Design	#Nets	No obfuscation PNR%	OER%	HD%	[60] PNR%	OER%	HD%	ΔWL%	ObfusX PNR%	OER%	HD%	ΔWL%	t_{CPU} (min)
c880	252	100.0	0.0	0.0	91.7	99.9	18.0	4.3	**85.3**	**100.0**	**23.3**	**3.4**	2.4
c2670	607	95.8	99.9	7.0	87.1	100.0	14.0	4.4	**77.8**	**100.0**	**23.5**	**3.2**	7.0
c3540	638	97.2	95.4	18.2	93.5	100.0	33.4	2.5	**84.5**	**100.0**	**38.2**	2.5	18.4
c5315	997	98.7	98.7	4.3	95.0	100.0	18.1	1.7	**88.9**	**100.0**	**23.2**	1.7	13.6
c6288	1921	99.8	36.8	3.0	98.6	100.0	42.1	1.8	**95.3**	**100.0**	**45.3**	1.8	14.1
c7552	1041	99.6	69.5	1.6	95.3	100.0	20.3	2.2	**87.5**	**100.0**	**27.2**	2.2	12.7
Avg.		98.5	66.7	5.7	93.5 (↓5.0)	100.0	24.3 (↑18.6)	2.8	**86.5 (↓12.0)**	**100.0**	**30.1 (↑24.4)**	**2.5**	11.4

of 0.5%, or is 3–5× more efficient in WL overhead for the same reduction of HR.

5.6.2 Wire Lifting with ObfusX

We show in Table 5.2 the performance and cost of wire lifting with ObfusX $(r = 5\,\mu\text{m})$ on ISCAS′85 benchmark designs, which are often used in related work, and compare them with [60]. The layouts are obtained from the authors of [60].

For this benchmark, we use the network flow attack model [59] which is obtained from the authors. Note that this is *not* a ML-based attack model and is *not* used to build ObfusX. Since the split layer for each design is not explicitly reported in [60], we tried to identify it by matching the number of nets on private layers with the number reported in [60]. ObfusX was applied on six designs for which we were able to identify the split layer, with WL budget equal to the reported WL overhead in [60]. The obfuscated layouts are converted to Verilog and their functional equivalency with original designs is verified with Synopsys Formality. For these designs, we use the following metrics to evaluate the performance and cost of an obfuscation.

- Percentage of netlist recovery (PNR) given in [43]: percentage of correctly reconstructed nets. This quantifies how well the attack can recover the whole design. *Lower is better.*
- Output error rate (OER): probability that there is any error bit in outputs of the reconstructed circuit. *Higher is better.*
- Hamming distance (HD) between outputs of the original and the reconstructed circuits. *Closer to 50% is better.*
- WL overhead (ΔWL%): percentage of increase in WL after the obfuscation. *Lower is better.*
- Total CPU time (t_{CPU}, in minutes) for running ObfusX.

We derive OER and HD from 100,000 runs of Monte Carlo simulations with ModelSim. OER and HD of the original design and [60], and the WL overhead of [60] are quoted from [60]. PNR of the original design and [60] are derived by definition, based on the design layouts and the reported numbers in [60].

As can be seen in Table 5.2, with reasonable computing time of 11 minutes on average, ObfusX reaches 100% for OER, and achieves better obfuscation in the reduction of PNR (12% vs 5% on average, or 2.4$\times$ better) and the increase in HD (24.4% vs 18.6% on average, or 31% better), with the same or less WL overhead compared to [60]. Note that the reported results of [60] come from a (best) combination of three obfuscation techniques including wire lifting and via perturbation for matching and non-matching v-pins, whereas in our results wire lifting is applied alone. In fact, our wire lifting and via perturbation techniques are orthogonal to each other. Therefore, they may be combined for potentially better performance.

We were not able to make a fair comparison with another related work [43] because the original layouts of [43] are likely to be very different from ours and were not made available. (The layouts in [43] are generated using all 10 metal layers, whereas our layouts from [60] only occupy 5–9 lower metal layers.)

In summary, for obfuscation with via perturbation, ObfusX is able to achieve a lower hit rate (indicating better obfuscation) while perturbing *significantly fewer* nets and vias in the design, with significantly lower wirelength. When the same wirelength limit is imposed during wire lifting, ObfusX performs significantly better in performance metrics (PNR and HD with equally good OER).

5.7 Conclusion

We presented ObfusX, a routing obfuscator for split manufacturing which incorporated SHAP-based analysis to explain a ML attack to the same problem. The unique benefits of ObfusX were in its ability to identify the best candidate nets for obfuscation, together with the layout features that make them most vulnerable when subjected to an attack. As a result, it achieved better performance than prior work while perturbing significantly fewer nets and with significantly lower wirelength during via perturbation. It also achieved significantly better performance than prior work if the same wirelength limit was imposed during wire lifting.

6 EXPLAINABLE PREDICTION OF DRC VIOLATION HOTSPOTS

Today's VLSI fabrication technologies require satisfying many complex design rules to ensure manufacturability. Creating a layout that is clean of design rule violations is now a cumbersome task, which may require many iterations in the design flow. Within the design flow, Design Rule Check (DRC) is typically applied after detailed routing. However, the process of detailed routing can be rather tedious and expensive, which typically takes several hours, if not days, to finish. Hence it is highly desirable that an inexpensive DRC predictor is developed so that DRC hotspots on the layout may be predicted accurately at the earlier stages in the design flow. In addition, it is beneficial if predictions for individual DRC hotspots can be properly *explained* in order to point to the root cause behind each individual violation. In this way, designers may leverage this early feedback without going through detailed routing and DRC phases each time.

6.1 Related Works on DRC Hotspot Prediction

Recent researches have focused on predicting routability and DRC hotspots [78, 8, 9, 54, 12, 55, 64] with ML. They have identified various features at the placement and/or global routing stages which can contribute to DRC violations. Several researchers [8, 9, 12] adopted support vector machines (SVMs) with radial basis function (RBF) kernels, Tabrizi *et al.* used a boosting ensemble model in [54] and a feedforward neural network (NN) in [55]. Xie *et al.* proposed RouteNet [64], a convolutional neural network (CNN) with transfer learning.

It is worth noting that, these works mainly focused on the predictive performance, without much consideration on other important aspects in practice, such as the model development cost and model explainability as well as data availability. Specifically, with a large number of features and samples, the SVM model with RBF kernel is expensive to train and the predictions are difficult to explain. As for data availability, some researches [54, 55] assumed that the accurate DRC results are partially available for random regions of the layouts; they split samples in the same design into training and testing sets based on this optimistic assumption. Moreover, unlike most other works, where each data sample contains layout information in a small region, RouteNet [64] used features in the entire layout as a single input to the model, and thus it requires hundreds of different placements from each design and the corresponding DRC results (after detailed routing each placement) for model training, assuming all macros are movable in placement. For this reason, data acquisition in [64] may be computationally expensive. Finally, no prior work provided model explainability to analyze each individual violations.

To address these issues, in this chapter, we propose to use the random forest (RF) classifier [5] as an ideal candidate to predict DRC hotspots, considering predictive performance and computational cost for model development. For the first time, we also provide consistent explanations for individual DRC hotspot predictions with SHAP [33], which works well with the RF classifier. Each explanation identifies top-ranked features and their amounts of contribution to a predicted DRC hotspot, which suggest the root causes behind each violation. Our contributions are summarized below.

- We propose model evaluation metrics that are tailored for DRC hotspot prediction where the number of DRC violations may be relatively very small, including area under the precision-recall curve for predictive performance, and number of predictive operations for model complexity.
- With these metrics, we carry out a comparative study on RF and ML

models from recent works on DRC hotspot prediction with similar settings [8, 9, 55, 12, 54], with standard procedures of cross validation and hyperparameter tuning, where data availability is carefully considered.

- By exploiting recent advances in explanatory analysis, we provide reasonable explanations for individual DRC hotspots predicted by RF, validated with real examples.

6.2 Overview of Explainable DRC Hotspot Prediction

In the DRC hotspot prediction problem, given a global routing (GR) outcome, we predict whether a global routing cell (g-cell), after detailed routing, will contain at least one DRC violation. The prediction is made based on other routed designs with the same technology and same design flow.

We show the overall workflow in Figure 6.1. Our approach is to formulate this problem as a supervised classification problem. We extract features from placement and GR (shown in the left panel of Figure 6.1) to form a feature vector (a.k.a. data sample) for each g-cell, and then feed it to a ML model, which accepts this feature vector as input and produces an output indicating how likely the g-cell is a DRC hotspot.

The data acquisition process is shown in the middle panel in Figure 6.1. We use 14 designs in 65 nm technology with five routing layers from the ISPD 2015 contest benchmark suite [6][1]. Each design is first fed into `Eh?Placer` [17], which produces a placed `.def` file. Then with Olympus-

[1]Design `edit_dist_a` is excluded from our experiments since it took more than 10 days to detail route and is therefore considered unroutable. `superblue` designs are excluded because the technology is different. We include two hidden designs `mult_2` and `mult_c` available at the contest website, which were released after the contest.

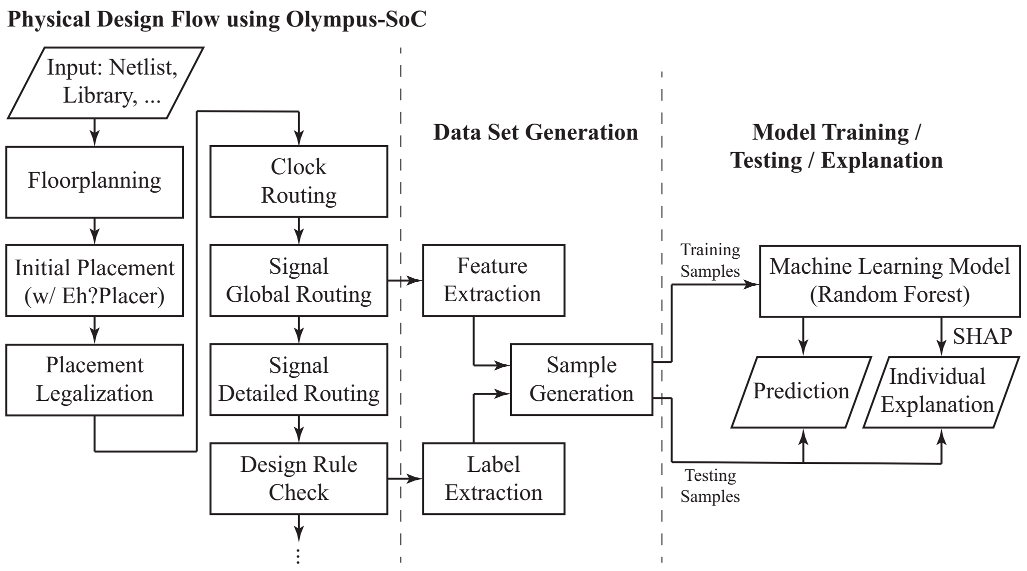


Figure 6.1: Workflow of explainable DRC hotspot prediction.

SoC, we follow a standard SoC flow with the steps shown in the figure. After GR of signal nets, the intermediate results are used to generate the feature vectors, and the DRC errors reported in the last step are used to determine whether they are actual DRC hotspots. Refer to Section 6.3.2 for details.

Model training and testing work as follows. The 14 designs are randomly divided into five groups with roughly equal number of samples, as shown in Table 6.1. For predicting the DRC hotspots in a specific design, we exclude the group that contains the design (i.e. testing group) and use all designs in the other four groups (i.e. training groups) for model training and tuning. The aforementioned procedure respects data availability by guaranteeing the *entire design* under test is never foreseen in the training stage. It avoids potential optimism in evaluation as in [54, 55], and better matches the practice in physical design, where the actual DRC errors become available after a design is detail-routed in its entirety.

Table 6.1: The Profile and Grouping of Designs

Design	# G-cells	# DRC hotspots	# Macros	# Cells (k)	Layout size (μm)
Group 1	**29994**	**364**	—	—	—
des_perf_b	10000	0	0	112.6	600×600
fft_2	3249	17	0	32.3	265×265
mult_1	8281	154	0	155.3	550×550
mult_2	8464	193	0	155.3	555×555
Group 2	**28263**	**547**	—	—	—
fft_b	6506	534	6	30.6	800×800
mult_a	21757	13	5	149.7	1500×1500
Group 3	**27826**	**669**	—	—	—
mult_b	24257	613	7	146.4	1500×1500
bridge32_a	3569	56	4	29.5	400×400
Group 4	**29689**	**738**	—	—	—
des_perf_1	5476	676	0	112.6	445×445
mult_c	24213	62	7	146.4	1500×1500
Group 5	**30318**	**298**	—	—	—
des_perf_a	11498	246	4	108.3	900×900
fft_1	1936	50	0	32.3	265×265
fft_a	6491	2	6	30.6	800×800
bridge32_b	10393	0	6	28.9	800×800

In the training stage of our workflow, we use grid search with 4-fold cross validation to find the best hyperparameters. This process consists of four passes. For each pass, designs in one of the four training groups are held out for validation and the designs in the remaining three groups are used for training. The performances from the four passes of cross validation are averaged. Then we compare and select the hyperparameter set with the best performance, and retrain the final model with the whole training set (i.e. all 4 training groups) before testing. As such, no *designs* (rather than samples) for validation is foreseen in the training process, which resembles the training-testing split and thus avoids the optimism bias in validation.

With predictions from the trained model, we explain them by inspecting the feature contributions to individual DRC hotspots with SHAP [33], as will be described in Section 6.3.4.

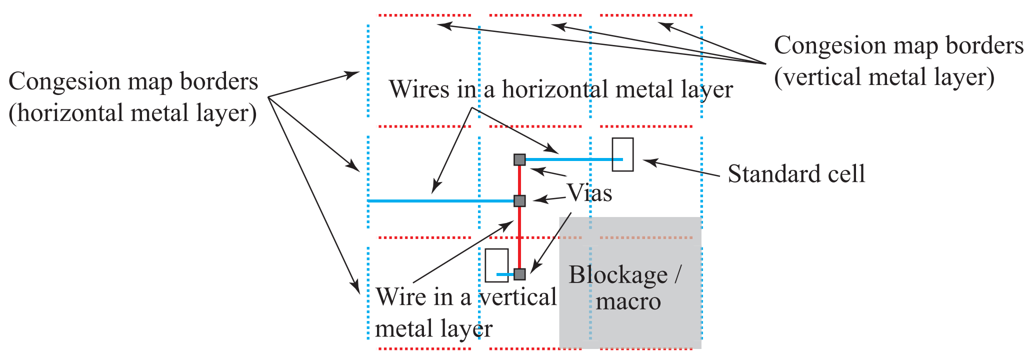


Figure 6.2: A 3×3 g-cell window with standard cells, wires (different colors indicate different metal layers), vias, congestion map borders, blockage/macro.

6.3 Details of Explainable DRC Hotspot Prediction

In this section, we elaborate our approaches to modeling predictions with Random Forest, defining features, labels, and metrics for evaluation, and providing explainability for individual predictions.

6.3.1 Random Forest and Its Benefits for DRC Hotspot Prediction

Random Forest (RF) classifier [5] is a well-known ensemble learning model based on decision trees, which typically trains hundreds of randomized decision trees and aggregates their outputs to generate a final prediction.

Compared to other classifiers with similar problem formulation for DRC hotspot prediction [8, 9, 54, 12, 55], RF has good and robust performance in prediction, owing to the ensemble mechanism. Moreover, because of the randomization in choosing the features to split, RF is robust in the presence of uninformative and redundant features. This property is especially good for our problem with a large number of features whose relative importance is not known beforehand.

It has relatively low computational cost because the training and testing of underlying decision trees are much more straightforward than optimization-based models like neural network, support vector machine, etc., as used in [55, 8, 12, 9]. Furthermore, the aggregation process is naturally good for parallelism, which further reduces the computational time where multiple computing cores are available. The low computational cost also makes it feasible to perform extensive searching for the optimal hyperparameters. Last but not least, the tree-based structure of RF makes it transparent in making decisions, providing good explainability, which we utilize to explain *individual* predictions, as elaborated in Section 6.3.4.

6.3.2 Feature and Label Extraction

Now we show the features and label as used in our predicting model. Figure 6.2 illustrates the types of layout information available after the placement and GR stages. Using these, prior works have defined different types of features related to routability and thus the DRC hotspot prediction, including

- Location of g-cells in the layout[54, 55],
- Density-related information (e.g., cell density [54, 55, 57, 9, 12], pin density [54, 57, 9, 12, 61, 78, 8], pin spacing/distribution [55, 9, 8]),
- Special pins and cells, which may have constraints in routing (e.g., clock pins [55], pins in nets with non-default rules (NDRs) [55, 57], and multi-height cells [9]),

- Connectivity, where complex connections around the g-cells may complicate routing (e.g., local nets [57, 9, 12, 78, 8], and cross-border nets [55, 8, 9, 12]),
- Congestion map [55, 57, 9, 12, 78], which indicates supply and demand of routing resources,
- Blockages for placement and/or routing [54, 55, 78], which further limit the routing resources.

Recent works [54, 55, 9, 12] also extract features in a window including neighboring g-cells to consider their contributions to DRC error due to potential routing detours.

Inspired by these prior works, in this chapter, we extract the following features in the designs, from the placed cells and the congestion map after signal GR, which are explained in Figure 6.2. Each data sample corresponds to a g-cell in the layout, which is expanded to a 3×3 *window* consisting of this g-cell (referred to as "central g-cell") and its 8 neighbors[2].

- For each of the nine g-cells in the window, we extract
  - The center x- and y-coordinates, normalized to $[0, 1]$.
  - The numbers of standard cells, pins, and clock pins that are fully inside the g-cell.
  - The number of local nets, defined as nets whose all pins are inside the same g-cell.
  - The number of pins that belong to any local net.
  - The number of pins that have NDRs, as defined in the ISPD 2015 contest benchmarks in our experiments.

[2]If the central g-cell is on the boundary of the layout, neighbors outside the layout are padded with blank g-cells.

  - The pin spacing, defined as the arithmetic mean of pair-wise Manhattan distances of pins inside g-cell.
  - The percentage of area occupied by blockages.
  - The percentage of area occupied by standard cells.

- For each of 12 congestion border edges (i.e., segments with blue/red dots in Figure 6.2) on *each metal layer*, and for each of 9 g-cells inside the window on *each via layer*,
  - The capacity C, defined as the maximum allowed number of wires/vias across the edge.
  - The load L, defined as the number of wires that are already across the edge (for metal layers) / the number of vias inside the g-cell (for via layers).
  - The resource margin, i.e., the difference of C and L.

We include almost all applicable features from prior works. This results in 387 features in total. To determine the labels, we examine the bounding boxes of DRC errors as reported by Olympus-SoC. A g-cell is a DRC hotspot if and only if the g-cell overlaps with any DRC error bounding box. A sample is positive if and only if the central g-cell is a DRC hotspot.

6.3.3 Metrics for Model Evaluation in DRC Hotspot Prediction

We propose several metrics for evaluating the predictive performance, complexity and computational cost of a model.

As can been seen in Table 6.1, DRC-violated g-cells are much fewer than DRC-free g-cells. Therefore, accuracy (i.e. the percentage of correctly predicted samples) is not a good indicator of model performance [19] for

DRC hotspot prediction. To address this, the following metrics are used in prior works.

- True positive rate $TPR = TP/(TP + FN)$, a.k.a. recall,
- False positive rate $FPR = FP/(TN + FP)$,
- Precision $Prec = TP/(TP + FP)$,

where TP and FP are the numbers of samples that are correctly and incorrectly predicted as positive, respectively; TN and FN are the numbers of samples that are correctly and incorrectly predicted as negative, respectively.

Although these metrics are better alternatives to accuracy, all of them can change when different thresholds of classification are applied [19]. Most works in DRC hotspot prediction [8, 9, 12, 55] used TPR and FPR only at a single threshold. In practice, however, the designer is free to adjust the threshold to get different prediction results with the same model. With this consideration, threshold-independent metrics, such as the areas under the receiver operating characteristic (ROC) curve and the precision-vs-recall (P-R) curve [19], are better indicators of overall model quality.

For the problem of DRC hotspot prediction in particular, one usually cares more about the performance when FPR is low (as the number of negative samples is typically large, a moderate FPR can imply a large number of undesired false alarms). For this reason, the area under ROC curve A_{roc}, which weights TPR over all FPRs equally, may not be effective enough. Therefore, we use the area under P-R curve (AUPRC) as the main performance metric, since it focuses more on the positive samples (both actual and predicted ones). For the same reason, AUPRC is used when we tune hyperparameters with cross validation. To understand how different models perform at a low FPR, we *also* report TPR and $Prec$ values at the classification threshold when $FPR = 0.5\%$. A similar FPR is seen in prior works [9, 64]. In summary, we use the following metrics for evaluation.

- TPR^*: true positive rate (a.k.a. recall) at the classification threshold such that $FPR = 0.5\%$.
- $Prec^*$: the precision at the same threshold as above.
- A_{prc}: the area under the precision-vs-recall curve.

6.3.4 Individual Explanations for Predicted DRC Hotspots

We use SHAP to analyze individual hotspot samples predicted by the RF model. With the help of SHAP values, we estimate how much each of the 387 features contributes to the DRC errors found at *individual* predicted DRC hotspots. Therefore, the designer is empowered to both predict and root cause individual DRC hotspots at an early design stage.

More specifically, the RF model maps each sample with feature vector $\mathbf{x} \in \mathbb{R}^n$, n being the number of features, to a probability $f(\mathbf{x}) \in [0, 1]$ indicating how likely the sample is a DRC hotspot. We adopt the SHAP tree explainer [33] to explain individual predictions made from RF, by looking into the most contributing features in specific DRC hotspots. We validate these explanations by comparing with actual DRC errors and detailed-routed layouts.

6.4 Experimental Results

We run experiments in a Linux desktop with an Intel 6-core 2.93 GHz CPU, an Nvidia 1080Ti GPU, and 24 GB memory.

6.4.1 Performance of DRC Hotspot Prediction

We compare the predictive performance of RF with all ML models applied in previous works that have similar problem formulations to ours

[8, 54, 9, 12, 55], including SVM with RBF kernel (SVM-RBF), random undersampling boosting (RUSBoost) with decision trees as base learners, and feedforward NNs. The inputs to all ML models are the 387 normalized features described in Section 6.3.2. Since we have no access to the exact post-route designs used in prior works, it is not possible to take their results directly. Instead, we implement each ML model with available information in these papers, train and tune the models with our best effort with cross validation, and evaluate with our dataset. All models are implemented using Python with scikit-learn (except for NNs: with Keras, backed by TensorFlow and GPU acceleration). We do not compare our work with [64] because the performances cannot be compared fairly due to different assumptions on macro movability in placement, as well as significant differences in the amount, scope, and acquisition cost of training data.

The performance is reported in Table 6.2 for 12 designs[3] in terms of TPR^*, $Prec^*$ and A_{prc}, as defined in Section 6.3.3. For example, the RF model shows an average TPR^* of 50.6%, meaning that RF predicts 50.6% of positive samples correctly on average while guaranteeing that $1 - 0.5\% = 99.5\%$ of negative samples are also predicted correctly.

Comparing the average performance in Table 6.2, we can find that RF is the best among all models in terms of all three performance metrics. Specifically, it is 21% better in A_{prc} than the popular SVM-RBF model on average and up to 60% better than other models. We highlight in bold the best performances among compared models for each design in Table 6.2. Then for each model and performance metric, we count the number of "winning designs" where the model performs the best among all models. Results show that RF wins the most designs in all three metrics, especially in the main performance metric A_{prc}.

Although the SVM-RBF model performs well in some designs, this

[3]Two other designs (`des_perf_b` and `pci_bridge_b`) are not included because there is no DRC errors and thus these metrics are undefined.

Table 6.2: Comparison of Performances of RF and Different ML Models Used in Prior Works

Design	SVM-RBF [8, 9, 12]			RUSBoost [54]			NN-1 [55]			NN-2			RF (this work)		
	TPR*	Prec*	A_{prc}	TPR*	Prec*	A_{prc}	TPR*	Prec*	A_{prc}	TPR*	Prec*	A_{prc}	TPR*	Prec*	A_{prc}
fft_2	**0.0588**	**0.0588**	**0.1358**	0.0000	0.0000	0.0183	0.0000	0.0000	0.0058	0.0000	0.0000	0.0062	0.0000	0.0000	0.0177
mult_1	**0.3961**	**0.5980**	**0.5248**	0.2143	0.4459	0.2944	0.2143	0.4459	0.3551	0.2857	0.5176	0.4132	0.3442	0.5638	0.4570
mult_2	**0.4767**	**0.6917**	0.5828	0.4041	0.6555	0.4798	0.1865	0.4675	0.3319	0.1658	0.4384	0.2930	0.4352	0.6720	**0.5845**
fft_b	0.1199	0.6809	0.4095	0.0206	0.2683	0.2816	0.0430	0.4340	0.1781	0.0375	0.4000	0.1752	**0.2416**	**0.8113**	**0.4404**
mult_a	0.6923	0.0763	0.2439	0.7692	0.0840	0.4435	0.1538	0.0180	0.0302	0.2308	0.0268	0.0259	**0.8462**	**0.0917**	**0.7445**
mult_b	**0.5498**	**0.7407**	0.6861	0.4633	0.7065	0.6324	0.4274	0.6895	0.5922	0.4356	0.6935	0.6134	0.5269	0.7324	**0.7025**
bridge32_a	**0.8393**	**0.7231**	**0.8703**	**0.8393**	**0.7231**	0.8418	0.7321	0.6949	0.7420	0.6786	0.6786	0.7246	0.7321	0.6949	0.8537
des_perf_1	0.5207	0.9362	0.8947	0.4127	0.9208	0.8129	0.3802	0.9146	0.8427	0.3846	0.9155	0.8488	**0.5459**	**0.9389**	**0.8954**
mult_c	0.5484	0.2194	0.1797	0.7258	0.2711	0.2929	0.6613	0.2531	0.3581	0.7903	0.2882	0.2792	**0.9032**	**0.3164**	**0.7180**
des_perf_a	0.5407	0.7037	0.7313	0.5366	0.7021	0.6740	0.5122	0.6923	0.6957	0.4878	0.6818	0.6309	**0.6748**	**0.7477**	**0.7797**
fft_1	0.1600	0.4706	0.3601	0.0600	0.2500	0.1261	0.0200	0.1000	0.1345	0.0800	0.3077	0.2016	**0.3200**	**0.6400**	**0.5348**
fft_a	**0.5000**	**0.0303**	0.0201	0.0000	0.0000	0.0050	0.0000	0.0000	0.0044	0.0000	0.0000	0.0098	**0.5000**	**0.0303**	**0.1009**
Average	0.4502	0.4941	0.4699	0.3705	0.4189	0.4086	0.2776	0.3925	0.3559	0.2981	0.4123	0.3519	**0.5058**	**0.5200**	**0.5691**
# Win. designs	6	6	3	1	1	0	0	0	0	0	0	0	**7**	**7**	**9**
# Model param.	1252.2k / model			318.5k / model			15.6k / model			15.9k / model			4269.7k / model		
# Prediction op.	3759.7k / sample			24.9k / sample			31.1k / sample			31.8k / sample			34.3k / sample		
Train. CPU time	65.7 min / model			6.9 min / model			24.4 min / model			13.1 min / model			8.9 min / model		
Pred. CPU time	0.24 min / design			0.03 min / design			0.01 min / design			0.01 min / design			0.04 min / design		

model has a fairly large number of parameters, the longest training time ($7\times$ more than RF), and the largest number of operations for a single prediction ($110\times$ more than RF). This is due to its high model complexity from the RBF kernel, hence the need to store many high dimensional support vectors as parameters and the complex calculations in both training and predicting. These facts also make it difficult to provide explainability with the SVM-RBF model.

The RUSBoost model has 100 iterations of boosting. It has the shortest training time and the fewest operations at prediction time owing to the simplicity of underlying decision trees. However, it is not very powerful in prediction, nor easy to parallelize due to sequential updates of model parameters.

We report two feedforward NN models with different number of hidden layers. NN-1 has a single hidden layer, which is the same architecture as in [55], except that 40 hidden neurons are used in our work as per cross validation. NN-2 has two hidden layers with 40 and 10 neurons, respectively. We use ReLU and sigmoid activations for hidden layers and the output, respectively. They are the simplest but also the least performing models among the five. In fact, we tried feedforwards NNs with more hidden layers with varying number of neurons and found that the predictive performance of feedforward NNs can hardly be improved further by adding more hidden layers beyond two. More complex and deeper NNs like convolutional NNs are not applicable to our dataset. Their complexity could also compromise explainability.

The reported RF model consists of 500 unpruned decision trees. That is why it has the most parameters among compared models. However, owing to the simplicity of decision trees, it does not require many operations in prediction with a negligible predicting runtime. The training time is also fairly short. Furthermore, unlike boosting methods, RF is easy to parallelize for training with more trees, which would not hurt the predicting performance as our cross validation suggests.

6.4.2 Explaining Individual Predictions

With the SHAP tree explainer in Python package `shap`, we measure feature contributions to individual RF predictions for hotspots and explain accordingly. Three typical DRC-violated g-cells in two designs are taken as examples in this experiment.

Figure 6.3(a)–(c) show the layouts of these example g-cells along with their neighboring cells, with red marks indicating the actual DRC errors (which are not available at prediction and explanation time). The cells, wires and vias are not shown. The colored edges show the GR edge congestion in layers M5 and M4 (a color closer to red indicates higher congestion). The via congestion is not visible. The gray box in (c) indicates a macro, where all wires and vias are blocked. The naming convention, which specifies the type, layer and location of GR congestion, is shown in Figure 6.3(d) to relate the feature names and the explanations below. As an example, feature `edM4_4V` is from the **e**dge congestion map. It is the **d**ifference of capacity and load in layer **M4**, on the edge labeled **4V**. Figure 6.3(d) shows more examples with corresponding colors.

Based on the RF model, the feature contributions to each hotspot prediction are evaluated and visualized with SHAP in Figure 6.4. Recall that the prediction output from the RF model is the probability that the sample is a DRC hotspot, and the threshold of classification is adjustable. Therefore a prediction output is not meaningful unless compared with that of other samples or the base value (i.e. the average). The pink/blue bars represent the positive/negative SHAP values of features, sorted by the absolute value. In other words, the pink/blue bars show how much each feature "pushes" the prediction output higher/lower (i.e. more/less likely to be DRC-violated) from the base value. Since these examples are actual DRC hotspots, the pink bars dominate the prediction and the blue bars

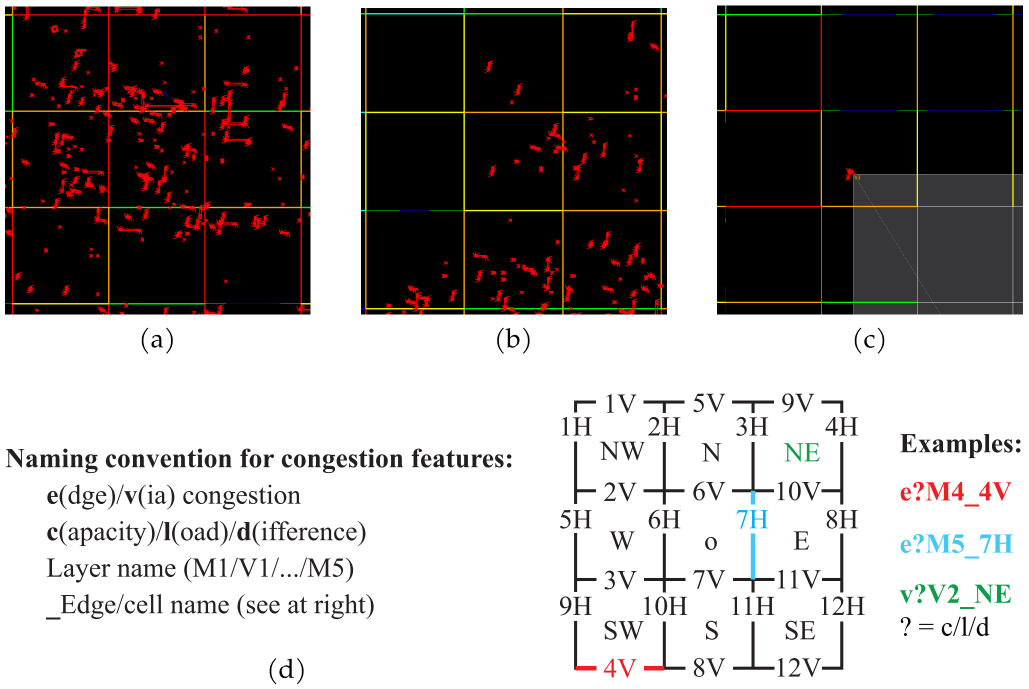


Figure 6.3: Example DRC hotspots to be explained. (a) A hotspot in highly congested area from `des_perf_1`. (b) A hotspot with moderate edge congestion from `des_perf_1`. (c) A hotspot near a macro from `matrix_mult_a`. (d) The naming convention for GR congestion features.

can hardly be seen. Referencing the naming convention in Figure 6.3(d), we can translate Figure 6.4 to the following explanations.

Figure 6.4(a) (corresponding to hotspot (a)) shows a lot of features pushing the prediction output from the base value 0.016 to 0.56, making hotspot (a) $35\times$ more likely to be a DRC hotspot than average. The most influential features are the GR edge overflows in layer M5. For example, the fact that `edM5_7H=-4` (i.e. the capacity of edge 7H in layer M5 is 4 tracks less than the load) pushes the output to the positive side by around 0.05, as indicated by the length of the rightmost pink bar. Similar overflows can be found at six other edges, which are all shown in red in Figure 6.3(a).

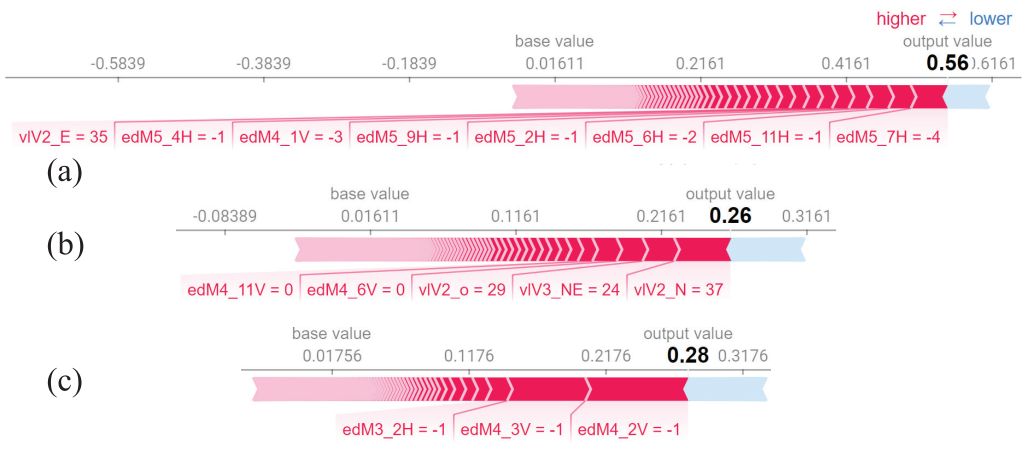


Figure 6.4: (a)–(c) Most contributing features for predicting DRC hotspots in Figure 6.3(a)–(c), evaluated by the SHAP tree explainer. (The blue regions on the right contain many features that are not visible as the pink bars dominate.)

Feature value `vlV2_E=35` is also shown in Figure 6.4(a), meaning the large load in layer V2 in the east neighboring cell also contributes to the DRC errors.

Hotspot (b) is mainly affected by the high via congestion in the north and the central cell in layers V2, as indicated by `vlV2_N=37` and `vlV2_o=29` in Figure 6.4(b), and in the northeast cell in layer V3 (`vlV3_NE=24`). The full loads in M4 on edges 6V and 11V (indicated by `edM4_6V=0` and `edM4_11V=0`, corresponding to the two horizontal orange edges in Figure 6.3(b)) have secondary contributions.

Hotspot (c) is primarily due to the three edge overflows in M4 (at edges 2V and 3V, the two horizontal red edges in Figure 6.3(c)) and M3 (at edge 2H). Other visible elements in the congestion map in Figure 6.3(c), such as the blockage of the macro and the overflow in M5 (indicated by the vertical red edge), are not the main reasons for this DRC hotspot.

These explanations are available individually on demand with runtime overhead of 1.4 sec/sample, without requiring actual detailed routing. To verify their validity, we compare them with the actual DRC errors of each example hotspot in Figure 6.3(a)–(c) after detailed routing, which are listed below.

(a) 60 errors of different types across metal layers M2 through M5 and via layers V2 through V4.

(b) Two shorts in M2, five end-of-line space errors (EOLs) in M3, two EOLs and a different-net space error in M4.

(c) One short in M3 and one short in M4.

All three explanations are consistent with the actual outcomes. For hotspot (a), a large number of overflows in the neighborhood pose extreme difficulty in routing. For hotspot (c), the explanation perfectly matches the layers where the errors are located. A closer look at the detailed wire and vias of hotspot (b) reveals that the main errors—EOLs in M3—arise due to the dense presence of vias in V2 and V3, which suggests the explanation to hotspot (b) also makes sense.

Notice that hotspots (a) and (b) have totally different explanations despite being from the same design and predicted by the same RF model. For hotspot (c), there is a GR edge overflow in M5 (vertical red edge in Figure 6.3(c)) that looks equally important as other overflows, but it is (*correctly*) not reported as a primary contributing feature. These facts suggest that the RF-based explainer does give reasonable, case-by-case explanations which can be beneficial at an early design stage.

6.5 Conclusion

We used RF to predict DRC hotspots at the global routing stage. In terms of AUPRC, a metric tailored for this purpose, RF showed up to 60% better predictive performance on average than ML models applied in similar works, with low computational cost. Owing to the transparency of RF and the adoption of SHAP, we can further make reasonable and consistent explanations to root cause individual DRC hotspots in an efficient manner. These facts make RF ideal for DRC hotspot prediction.

7 APPROXIMATE LOGIC SYNTHESIS GUIDED BY EXPLAINABLE ML

Approximate logic synthesis (ALS) is the process of generating a Boolean circuit that approximates the functionality of an original circuit within a tolerance of error, in trade of better quality of results (usually a smaller area, power and/or delay). Many ALS techniques have been proposed in the past decades [51]. Depending on the abstract level the approximation is performed, these techniques can be broadly classified into two categories. Some techniques are based on manipulating the circuit structure/netlist (e.g., [56]). The other category of ALS techniques are based on relaxation of the functionality, essentially by altering entries in the truth table, regardless of the circuit structure.

As a specific subset of the latter category, some ALS techniques (e.g., [40]) construct an approximate circuit only based on samples of input-output pairs (i.e., entries in a truth table), which are generally achieved by learning a function that generalizes these samples. We refer to this subset of techniques as *sampling-based ALS* in this dissertation. Sampling-based ALS is drawing increasing research interest, as many promising studies and interesting contests are conducted in recent years [10, 13, 4, 52, 45].

Sampling-based ALS techniques are natural fits for synthesis of ML applications, where only samples corresponding to a training set are known and trading accuracy with the implementation cost is acceptable [16]. However, they can also be useful in other application domains, and may be integrated with conventional logic synthesis and optimization techniques. For example, an approximated circuit may first be constructed from the provided samples using sampling-based ALS, and serve as a starting point to apply conventional synthesis and optimization techniques to further optimize it.

Moreover, sampling-based ALS is also a natural fit when the input space is known to be a constrained version of all possible inputs. For example, if a designer knows in advance that some combinations of input bits cannot happen in their specific modes of operation, then these constraints can be easily incorporated in sampling-based ALS; if a set of samples representing all modes of operation is given, then one can easily prune the undesired samples and only use the ones corresponding to specific modes of operation, keeping the sampling-based ALS process intact.

Recent CAD contests (ICCAD 2019 and IWLS 2020) have introduced related problems on synthesis only based on samples of input-output pairs. The ICCAD 2019 contest [52] required an almost-perfect accuracy from synthesis and it did not target ALS specifically. The IWLS 2020 contest [45] looked at a rather restricted sampling-based ALS problem where the size of the training set was very small and a limit of 5000 gates was imposed for synthesis of all functions regardless of number of their primary inputs.

This chapter studies sampling-based ALS problem using adaptive decision trees (ADTs) with specific focus on utilizing explainable ML. We formulate the approximation of a Boolean function as a supervised ML problem, and propose to use explainable ML to guide the training of ADTs. This is based on a feature importance metric derived from SHAP [34]. With SHAP-powered explainability, we measure the importance of each individual primary input bit with respect to the function output, which we further utilize to achieve an efficient ADT-based implementation of Boolean function approximation with minimal loss in accuracy. We also include approximation techniques for ADT which are specifically designed for ALS, including don't-care (DC) bit assertion and instantiation.

ADT learns a generalized sum-of-product (SoP) expression of the underlying Boolean function from sample input-output pairs. It is based on recursively applying Shannon expansion of the underlying function with splits on a series of primary input bits. The selection of these input bits are guided by a SHAP-based bit importance metric, which is more consistent and beneficial for ALS purposes than other previously adopted metrics, such as the decrease in impurity and in entropy (aka. information gain) [40, 4].

In our experiments, we use classic synthesis tools (`espresso` and `abc`) to achieve a baseline and accurate implementation (i.e., no error). We further restrict the input space and show the same synthesis flow only results in 4% reduction in area on average when no approximation is applied. However, applying approximation with the proposed SHAP-guided ADT techniques results in 39%–42% reduction in area with 0.20%–0.22% error on average.

In the remainder of the chapter, we review some preliminaries in Section 7.1 and related work in Section 7.2. We discuss our approximation techniques in Sections 7.3 and 7.4, followed by presentation of experimental results in Section 7.5 and conclusions.

7.1 Preliminaries on Sampling-Based ALS

In this section, we briefly introduce the technical backgrounds of sampling-based ALS, reformulated as a supervised ML problem.

7.1.1 Binary Decision Tree for Learning Boolean Functions

A decision tree classifier [21, 44] is a ML model that iteratively splits training samples into subsets (branches) based on the value of a selected feature, forming a tree-like structure that is used to classify any other samples. The selection of splitting features is typically based on Gini impurity or entropy. Specifically, in the context of Boolean functions, a training sample corresponds to an input vector and its corresponding output. Each feature corresponds to an input bit (which means the feature can only take a value of either 0 or 1) and the class label of a sample is determined by the output bit (0 or 1).[1] This binarized classifier is called a binary decision tree (BDT). Prior studies have shown its efficacy on learning Boolean functions [45, 13]. The tree structure, which resembles a binary decision diagram (BDD), makes it straightforward to convert the learned function to a truth table for logic synthesis.

7.1.2 Adaptive Decision Tree

A conventional decision tree algorithm accepts a single batch of training samples, which are split into new branches as the tree develops. As the tree grows deeper, the number of samples in each branch becomes exponentially smaller. Therefore, it is likely to overfit locally. To address this, each time a tree node is split into two nodes, a new batch of samples (of the original size) are used for each new node for further tree development [13]. In this chapter, we refer to this method as Adaptive Decision Tree (ADT) since new samples are generated and consumed "adaptively" on the fly as the tree grows. While not very common in conventional ML problems due to limited data availability, ADT is feasible in the context of ALS since we can sample the input space and get as many input-output pairs as needed. Details of ADT will be discussed in Section 7.3.2.

7.2 Related Works on Sampling-Based ALS

The study of learning functions from sample input-output pairs remains active in recent research and contests [10, 45, 52, 13, 4, 20]. Chatterjee [10] explore an interesting idea of learning a Boolean function and synthesize it as a network of lookup tables (LUTs) by simply memorizing the samples. Boroumand et al. [4] generate Boolean networks by building LUTs that

[1]We use "feature" and "(input) bit" interchangeably in this chapter.

are most informative in terms of decrease in entropy (aka. information gain). Targeting the same problem as in this chapter, the technique of these two papers is specific to building the approximated circuit with LUTs. In contrast, our approach provides the approximated AIG, which offers more flexibility in later synthesis steps (logic optimization, technology mapping, etc.).

Recent contests suggest research interest in related topics. Specifically, ICCAD 2019 contest [52] focuses on reconstructing the logic function inside a black-box circuit given full access to the circuit. IWLS 2020 contest [45] explores the possibility of learning Boolean functions with a very small number of samples and synthesizes them as AIG within a gate count limit. Both contests did not specifically target ALS; the ICCAD contest problem targeted an extremely high accuracy of 99.99% for application domains when such high accuracy is needed such as security. The IWLS contest defined a restricted problem where a limit of 5000 gates was imposed for synthesis of all the functions regardless of the size of primary inputs, with the objective to maximize accuracy.

The closest related works to ours were [13] and [20]. The idea of ADT was adopted in [13] as part of techniques to tackle the ICCAD 2019 contest problem. However, our work distinguishes from [13] in three aspects. First, [13] focused on solving the ICCAD contest problem and not ALS; it focused on synthesis with an extremely high accuracy of 99.99%. Second, the splitting criterion of ADT in our work is different and is based on ML explainability. Third, we further analyzed the error bound and addressed the importance of DC/XOR bit assertion and sample enumeration, which are specially designed for ALS. In [20], the authors used a conventional DT to perform logic optimization. They compared the learned SOP expression with the output of espresso, and conclude that a DT with limited depth is able to learn expressions that can be synthesized to more size-efficient circuits. However, they only targeted functions with less than 16 primary inputs. Our experiments will show that conventional DTs, either regularized or not, is not suitable for functions with a larger number of primary inputs. Finally, neither [13] or [20] addressed the tree complexity introduced by possible XOR gates and therefore could take a long time to build the tree in case of a large tree size.

7.3 Overview of SHAP-Guided Logic Approximation

In this chapter, we consider the following sampling-based ALS problem. Given a Boolean function $f : \{0,1\}^n \to \{0,1,-\}$ that is specified by a truth table with unknown circuit structure, where $-$ denotes a don't-care (DC) output, the goal is to find an approximate circuit that implements $f' : \{0,1\}^n \to \{0,1\}$ with a small area, where the output error rate $\mathrm{Prob}_{\mathbf{x}\in\{0,1\}^n}(f(\mathbf{x}) \neq f'(\mathbf{x}) \mid f(\mathbf{x}) \in \{0,1\})$ does not exceed an error bound ε. In this work, the circuit is implemented as an And-Inverter Graph (AIG) according to the flow that will be explained next. Other synthesis and optimization techniques, such as logic rewriting and technology mapping, may be applied thereafter to the produced AIG.

7.3.1 Overall Workflow

We first discuss the overall flow of ALS. Figure 7.2 shows three cases corresponding to different target requirements. The baseline flow shown in Figure 7.2(a) is an example of synthesizing from a truth table (in PLA format) without approximation, and when the input space is not constrained (i.e., all combinations of values of primary inputs are feasible in the target application). The truth table is first fed into `espresso`, a two-level logic minimizer. Then the minimized PLA is provided to `abc` for synthesis. The output of the flow is an AIG, which is used in further synthesis steps such as technology mapping. As shown in Figure 7.2(b), when the input space is constrained, the constraints can be added in PLA as DC cubes. Note that as a two-level logic minimizer, `espresso` is not optimized for multi-level logic synthesis. This means adding DC cubes may or may not result in smaller area in AIG processed with it. Therefore, as shown in Figure 7.2(b), we accept the constrained-input-space version only if the area is improved, hence the "choose one" step. However, this issue is beyond the scope of this work. Figure 7.2(c) includes the proposed module of ADT-based approximation (highlighted in red), which is the core idea that will be discussed next.

7.3.2 Adaptive Decision Tree for ALS

Since we have complete information on the original function, but no information on the circuit structure, our approach to sampling-based ALS is to formulate it as a supervised learning problem, where an approximated Boolean function is learned based on input-output pairs sampled from the function. We adopt adaptive decision tree (ADT) as our learning model as reviewed in Sections 7.1.1 and 7.1.2. It accepts the truth table as input, and outputs an approximated truth table.

ADT learns an approximated function by dividing the input space $\{0, 1\}^n$ iteratively based on the following formula known as *Shannon expansion*.

$$f(x_1, \ldots, x_i, \ldots, x_n) = (x_i \wedge f|_{x_i=1}) \vee (\neg x_i \wedge f|_{x_i=0}), \tag{7.1}$$

where $f|_{x_i=1} = f(x_1, \ldots, 1, \ldots, x_n)$ is a subfunction of f with x_i assigned to 1, and $f|_{x_i=0}$ is similarly defined. They are referred to as the *cofactors* of f.

When building an ADT, we maintain a list of bit states in each tree node. Each bit has one of the five states:

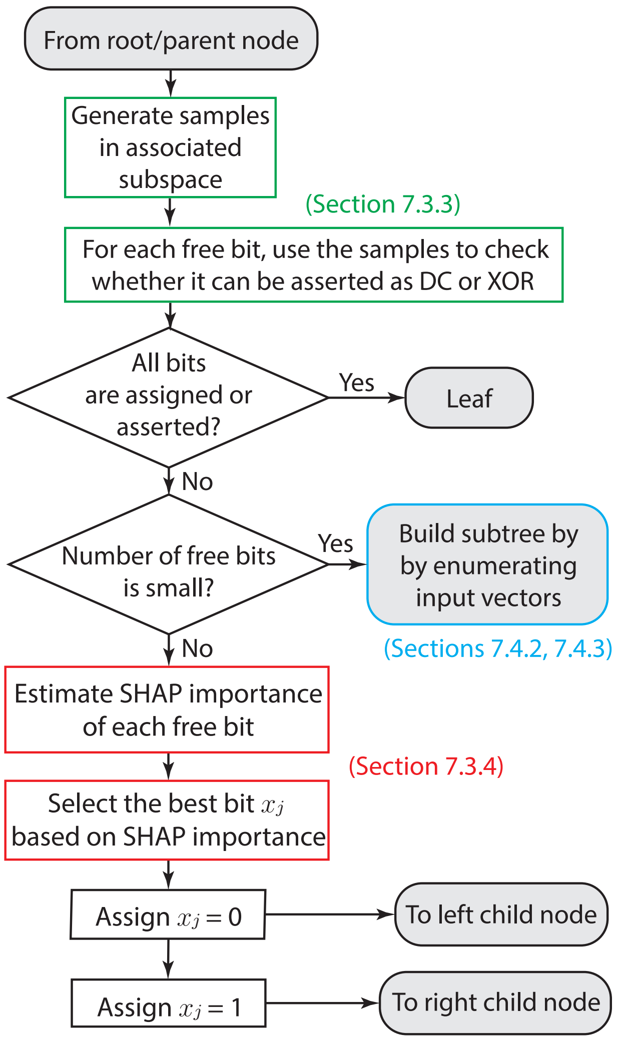

From root/parent node
Generate samples in associated subspace
(Section 7.3.3)
For each free bit, use the samples to check whether it can be asserted as DC or XOR
All bits are assigned or asserted?
Yes
Leaf
No
Number of free bits is small?
Yes
Build subtree by by enumerating input vectors
(Sections 7.4.2, 7.4.3)
No
Estimate SHAP importance of each free bit
(Section 7.3.4)
Select the best bit x_j based on SHAP importance
Assign $x_j = 0$
To left child node
Assign $x_j = 1$
To right child node

Figure 7.1: Illustration of building an ADT for ALS using SHAP importance. Steps taken to process a node are illustrated which could be to further split the tree, reach a leaf, or build a subtree at one shot.

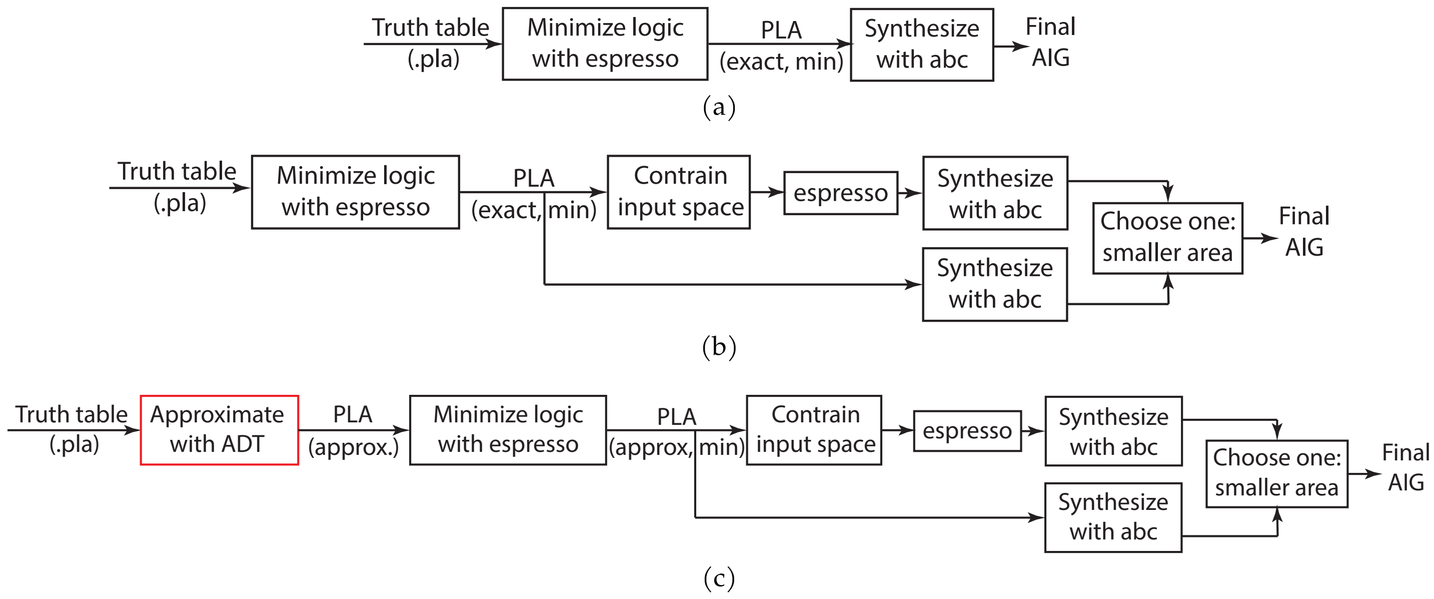


Figure 7.2: Logic synthesis flows compared: (a) Exact functionality without constraint in input space (baseline), (b) exact functionality with constraints in input space, and (c) approximated functionality with constraints in input space (proposed). Other synthesis techniques may be applied thereafter to the produced AIGs.

0 : assigned to 0,

1 : assigned to 1,

$-$: *asserted* (approximated) as DC bit, details in Section 7.3.3,

$\times$: *asserted* (approximated) as XOR bit, details in Section 7.3.3,

$?$: free, i.e. yet to explore.

The building of ADT begins with the root node, where all bits are initially in free state. At each node, as illustrated in Figure 7.1, we first generate samples that are consistent to the states of input bits: bits which are already set to 0-state (resp. 1-state) must be set to 0 (resp. 1). Then we visit each free bit and use the generated samples to evaluate if that bit can be asserted (i.e., approximated) as a DC or XOR bit[2]. A bit will be asserted if found to be feasible after evaluation, and otherwise will remain free. Once all bits are visited, we select a free bit indexed j as the next node to split. (This selection of the next bit is SHAP-guided to choose the most suitable bit to split on.)

We split the node into two child nodes with assignments $x_j = 0$ and $x_j = 1$ respectively. The state of bit x_j in the two child nodes is respectively set to 0 and 1. This means when we proceed to the left (resp. right) child node and generate samples, x_j has to be fixed to 0 (resp. 1). The child nodes represent its two cofactors $f|_{x_j=0}$ and $f|_{x_j=1}$ in (7.1) respectively. Equivalently, each child node focus on the output of f in an input subspace where an input bit is fixed. We will find these cofactors in corresponding child node and do it recursively, until each node in the tree either splits or is a leaf. The ADT can be built in either depth-first or breadth-first order. We opt for breadth-first order for more balanced development in each branch.

[2]XOR bit is opposite of a DC bit and defined in Section 7.3.3

As illustrated in Figure 7.1, when building an ADT, each tree node has one of three outcomes: (a) it is a leaf node, making no further split on this branch, (b) it splits into two child nodes with assignments on an input bit x_j, or (c) it builds a subtree at one shot if the number of free bits is small (details in Section 7.4.2). The outcome depends on a groups of input vectors sampled in the associated subspace.

Recall that building an ADT is essentially recursive Shannon expansions with (7.1). When the building process concludes, we will get a sum of product (SoP) expression of f.

The reason why ADT yields an *approximated* version of f is that we **assert** DC and XOR bits only based on samples. For example, if the output is the same for all samples in an input subspace, we assert that the function output is constant in this subspace. In this case, all free bits will be set to DC ("$-$") state. Note that due to statistical sampling error, a bit may be asserted DC/XOR when it is actually not, but very close. This is the core that we base our sampling-based ALS approach on, which is discussed in detail next.

7.3.3 Assertion of DC Bits and XOR Bits

For combinational circuits built with logic gates, it is possible that an output bit does not depend on certain primary inputs, especially in an input subspace. These input bits are referred to as don't care (DC) bits. For example, for function $f(x_1, x_2, \ldots, x_n) = x_1 \vee g(x_2, \ldots, x_n)$, f does not depend on $x_2, x_3, \ldots$ in the subspace where $x_1 = 1$. Therefore, it is desired to assert them as DC in this subspace.

An XOR bit is the opposite of a DC bit. When an XOR bit flips and other bits remains the same, the output flips regardless of other input bits. For example, in function $f(x_1, x_2, \ldots, x_n) = x_1 \oplus g(x_2, ..., x_n)$ where g is an arbitrary function, x_1 is an XOR bit of f.

The assertion of XOR bits in ADT is important. Generally, we want to *avoid* splitting the tree on any XOR bit, as it would create two identical subtrees with opposite polarities. As illustrated in Figure 7.3, splits only happen on free (not-assigned/asserted) bits. Preliminary experiments showed this strategy can save about 50% of the runtime to build the tree when the circuit includes many XOR gates.

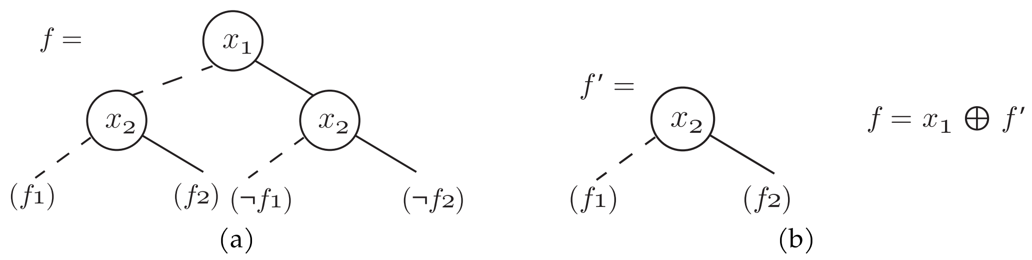


Figure 7.3: XOR bit assertion in ADT for $f = x_1 \oplus ((\neg x_2 \wedge f_1) \vee (x_2 \wedge f_2))$ where f_1 and f_2 are Boolean functions independent of both x_1 and x_2. Dash lines are 0 assignments and solid lines are 1 assignments. (a) Not asserting x_1 as an XOR bit and instead splitting on x_1 would result in two subtrees with opposite polarities. (b) Asserting x_1 as an XOR bit, we only need to develop one subtree and take the XOR of x_1 and the resulting function f'.

7.3.3.1 Asserting DC and XOR bits by sampling

As shown in Figure 7.1, we use generated samples at each node and visit each free bit to determine if it can be asserted to DC or XOR. To determine if a bit x_j can be asserted to DC in a subspace, we generate $2r$ samples as r pairs. Each pair consists of two random samples in the subspace which only differs in x_j. We compare the two outputs of each pair of input samples. The DC assertion fails if any pair has different outputs, otherwise x_j is asserted as DC after comparing all r pairs of outputs.

The assertion of an XOR bit is similar. When we compare the output in pairs, the outputs of each pair are expected to be different. The XOR assertion fails if any pair has the same outputs, otherwise x_j is asserted as an XOR bit after comparing all r pairs of outputs. The assertion step is repeated for every free bit in a node. A total of $2rm$ samples will be generated, where m is the number of free bits.

7.3.3.2 Expected error rate of a fully developed ADT

Now we analyze the error of our ADT-based ALS. Consider the assertion on bit x_j in the root node of an ADT. Assume for p-fraction of input vectors $\mathrm{x} \in \{0,1\}^n$, a single flip in bit x_j results in a change in the output. With r pairs of random samples of x generated as previously described, the probability of a bit being incorrectly asserted as DC or XOR is

$$P(p,r) = p^r + (1-p)^r \approx (1-p)^r. \tag{7.2}$$

The above approximation holds when p is close to 0. If p is not close to 0 or 1, $P(p,r)$ becomes exponentially small with growing r and is negligible. If p is close to 1, a similar conclusion can be drawn by symmetry. Without loss of generality, we assume p is very close to 0, meaning for most x in the input subspace, a flip in bit x_j will not change the output. In case of incorrectly asserting x_j as a DC bit, the error rate caused by this assertion will be $p/2$ because it is considered an error only when the function outputs differently for $x_j = 0$ and $x_j = 1$, and only one of them causes the error. Therefore the expected error rate due to incorrectly asserting a bit is

$$\varepsilon_0 \approx \frac{p(1-p)^r}{2} \leq \frac{1}{2(r+1)}\left(1 - \frac{1}{r+1}\right)^r \approx \frac{\mathrm{e}^{-1}}{2r}.$$

This bound for the expected error also holds for any other nodes, except that nodes in level k (i.e., k levels below the root, where k variables have been assigned) would cause an expected error rate of $p/2^{k+1}$ in case of incorrect assertion, since each node in level k only represent $1/2^k$ of the input space. Correspondingly, there are at most 2^k nodes in level k, which cancels out the factor of $1/2^k$ at each node. Therefore, the total expected error rate is bounded by $\mathrm{e}^{-1}/2r$ for each incorrect bit assertion in a specific level of ADT.[3] Since each bit can be asserted at most once along any path from the root to a leaf, the expected error is

$$\varepsilon \leq n\varepsilon_0 \leq \frac{\mathrm{e}^{-1}n}{2r} \tag{7.3}$$

for the entire ADT. For $n = 50$, $r = 1000$, this bound evaluates to 0.9%.

Note that is the bound for the *expected* error and is not guaranteed due to randomness. However, it can be useful as a quick estimate and as a guideline to determine the number of samples r for bit assertion given a target error rate.

7.3.4 SHAP-Importance-Based Splits

The next question to consider is, if there are free bits in a tree node, how to select the best free bit to split at each node of ADT. Conventionally in a decision tree algorithm, samples in a node is split on a feature that will maximally reduce the impurity [21] or entropy [44] of the node. However, in the context of ALS, either impurity or entropy is not always a good metric. Alternatively, we propose to split based on the SHAP importance, as defined in (4.6), which can be interpreted as an overall importance of an input bit with respect to the output bit.

In the context of ALS, SHAP importance of a primary input bit measures its contribution to the primary output in the approximated circuit. In other words, how much a change in the output of a Boolean function can be attributed to each input bit. The SHAP importance relates to the nature of Boolean functions with the following propositions.

[3]This bound is likely to be very loose because the number of nodes is usually far less than 2^k as we go deep in the tree.

Proposition 7.1. *The SHAP importance of any input bit x_j in a Boolean function f is in the range of* $[0, 0.5]$.

Proposition 7.2. *The SHAP importance of bit x_j in f is 0 if and only if f is independent of x_j.*

Proposition 7.3. *The SHAP importance of bit x_j in f is 0.5 if and only if f solely depends on x_j, i.e., $f = x_j$ or $f = \neg x_j$.*

We show with minimal examples why SHAP-importance-based splits is good for ALS, especially superior to impurity- or entropy-based splits. Consider function $f_0(x_1, x_2, x_3) = x_1 \oplus x_2$, where $\oplus$ is XOR. Intuitively, x_1 and x_2 should be more important than x_3 because flipping either x_1 or x_2 would always change the output while x_3 is irrelevant. However, an impurity- or entropy-based decision tree makes no split (or a completely random split, depending on training options) because splitting on any input bit would not change the impurity/entropy. In fact, the above conclusion can be generalized for any function $f^*(x_1, x_2, x_3, \dots, x_n) = x_1 \oplus x_2 \oplus g(x_3, \dots, x_n)$ where g is an arbitrary Boolean function. In this case, we refer to x_1 and x_2 as *XOR bits*. Table 7.1 shows the change of impurity and entropy when splitting on different input bits and the SHAP importance of each variable. For each metric, the bit with the maximum value in the table will be selected for the split. From Table 7.1, impurity- or entropy-based decision trees would make no split or completely random splits for any function f^*, whereas SHAP importance can differentiate the XOR bits x_1 and x_2 from other input bits.

In [13] where ADT is adopted for logic reconstruction, the authors propose another bit importance metric—the absolute difference of its two cofactors

$$Q'_j = \mathbb{E}(f|_{x_j=0} \oplus f|_{x_j=1}). \tag{7.4}$$

We notice that although computationally more efficient, it is inferior to SHAP importance. Consider function $f = x_1 \wedge (x_2 \oplus x_3)$, where x_1 is

Table 7.1: Comparison of different metrics as splitting criteria

Metric	x_1 or x_2	x_3 in f_0	$x_j, j \geq 3$ in any f^*
Decrease in impurity	0	0	0
Decrease in entropy	0	0	0
SHAP importance Q_j	$\in [0, 0.5]$	0	$0 \leq Q_j \leq Q_1 = Q_2$

intuitively more important than the other two, while x_2 and x_3 are equally important. However, the proposed metric in [13] $Q' = (\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$ suggests all three bits are equally important, whereas SHAP importance $Q = (\frac{1}{4}, \frac{1}{8}, \frac{1}{8})$ indicates that x_1 is more important than x_2 and x_3, which matches well with intuition.

These examples motivate us to explore ADTs with splits based on SHAP importance. We propose to split on the bit with maximum SHAP importance. This is done after asserting any XOR bits.

Recall that exact evaluation of SHAP importance is not efficient. In fact, it is not necessary since we are only interested in one input bit with the maximum SHAP importance. Therefore, we use the following two methods to estimate the SHAP importance.

7.3.4.1 Monte Carlo estimation

By extending and rearranging the formula of SHAP value in (4.4) in the context of Boolean function f, we can rewrite it as

$$c(\mathbf{x}, j) = \sum_{\mathbf{z} \in \{0,1\}^n} w(\mathbf{z}, \mathbf{x}, j) \left[f(\mathbf{z}|_{z_j = x_j}) - f(\mathbf{z}|_{z_j = \neg x_j}) \right], \tag{7.5}$$

where $\mathbf{z}|_{z_j = x_j}$ is vector $\mathbf{z}$ whose j-th bit is substituted by x_j. $w(\mathbf{z}, \mathbf{x}, j)$ is the weighting factor given by

$$w(\mathbf{z}, \mathbf{x}, j) = \frac{1}{2n} \sum_{k=d}^{n-1} \frac{1}{2^k} \binom{k}{d}, \text{ where } d = \sum_{i \neq j} 1(z_i \neq x_i). \tag{7.6}$$

Note that the weight only depends on d, the number of different bits in $\mathbf{z}$ and $\mathbf{x}$ excluding bit j. The maximum weight occur when $d = 0$, and as d increases the weight decays exponentially. Exploiting this fact, we can efficiently estimate the SHAP importance of bit j by its first-order approximation (i.e., only consider the term with $d = 0$) from a group of carefully generated samples according to (7.5).

Since the weight is constant when $d = 0$, we drop it for simplicity; we also have $\mathbf{z} = \mathbf{x}$ with possible exception for bit x_j. The SHAP importance is therefore approximated as

$$Q_j \approx \mathbb{E}_{\mathbf{x}} \left| f(\mathbf{x}|_{x_j=1}) - f(\mathbf{x}|_{x_j=0}) \right|, \tag{7.7}$$

Equation (7.7) can be estimated by Monte Carlo (MC) simulation. Recall that we generated samples in the step of DC/XOR bit assertion. These samples can actually be reused to estimate (7.7) owing to the same pattern of sample generation for both tasks.

7.3.4.2 Tree SHAP estimation

Although the first-order approximation of SHAP importance (7.7) is computationally efficient, its formulation coincides with (7.4), the bit importance metric adopted in [13]. As discussed in Section 7.3.4, it is not as good as the original metric and sometime may deviate much from (4.6). A more accurate method of estimating (4.6) is to adopt the exact evaluation of SHAP values specifically optimized for tree-based models, known as *Tree SHAP* [33]. To utilize it, we train a temporary decision tree by reusing all samples generated in DC/XOR bit assertion step in Section 7.3.3 (i.e., based on a total of $2rm$ samples, where m is the number of free bits in the node before the assertion step). This temporary tree is only used for Tree SHAP evaluation and is discarded thereafter. For each free bit j, we use Tree SHAP to evaluate its SHAP value $c(\mathbf{x}^*, j)$ based on one of $2rm$

samples $\mathbf{x}^*$, and average the absolute value of SHAP value over these samples to get the SHAP importance Q_j with (4.6).

7.4 Implementation Details

7.4.1 Number of Samples for Each Split

As discussed in Section 7.3.3, the expected error decreases with r, therefore it is good to control the number of samples for the trade off between error and runtime. Each function is different and therefore the tightness of the bound in (7.3) may vary. Empirically, a value of r should be tried first, and if the error is not satisfying and runtime permits, we can try larger r values for potentially better error and/or area. Empirically, the error drops by half when r is doubled, which is consistent with (7.3). In our experiments, we begin with $r = 128$, which yields an error rate up to 3%, and we keep doubling the r values up to $r = 2048$.

7.4.2 Sample Enumeration

As discussed in Section 7.3.3, at each node of the tree, we generate $2rm$ random samples to check whether any of the m free bits can be asserted. We notice that, when m is small, it is more efficient to enumerate all 2^m input samples than sampling. This happens when $2^m \leq 2rm$. When $r = 2048$, for example, this inequality solves to $m \leq 16$. In that case, we do not use the SHAP based bit importance to split the node one bit at a time. Instead, we build a *subtree* for the node at one shot, based on all 2^m samples.

To build a subtree, we use (7.7) to determine the splits for its simplicity. However, we do not generate new samples as in the generate flow. The building of subtree concludes when each leaf node contains only one input sample, or multiple samples sharing the the same output.

7.4.3 Instantiation of DC Bits

As discussed above, when building the subtree with sample enumeration, we will generate 2^m samples where m is the number *free* bits. When we enumerate these samples, each previously asserted DC bits has to take a value of either 0 or 1; we call this process the *instantiation* of DC bits.

At first thought, it should not matter which value to take for DC bits because, by definition, the output will be the same. However, it is not the case because we may have asserted some DC bits incorrectly. In that case, if we instantiate DC bits randomly, the outputs could be mixed and random, and therefore the resulting subtree can be complex and the synthesized area could be large. This is essentially the same overfitting issue as training with a conventional DT, as we will see with the experiments in Section 7.5.

To address this issue, we propose to select a fixed default vector $\mathbf{x}^{(0)} \in \{0,1\}^n$ throughout the entire ADT building process. Each time a subtree needs to be built, the DC bits in any enumerated sample $\mathbf{x}$ are instantiated with corresponding bits in the default vector, i.e.,

$$x_j = x_j^{(0)} \text{ for each DC bit } j.$$

For example, if the default vector is set to all zeros, all DC bits should be set to zero when enumerating samples. Note that this is applied only in sample enumeration for subtree building (the blue block in Figure 7.1), which is different from sample generation (the green block in Figure 7.1) where previously asserted DC bits can take either value. In our experiments, we set a random default vector $\mathbf{x}^{(0)}$ for each function.

7.4.4 Constraints in Input Space

A unique feature of sampling-based ALS is that it decouples the approximation and synthesis parts of the flow, making the approximation models (e.g., ADT) orthogonal to logic synthesis and optimization methods (e.g., `espresso`, `abc`) and potentially other ALS techniques. As a result, it can straightforwardly handle the function with a non-empty DC set, or equivalently, the case where the input space is constrained by Boolean conditions (e.g., $x_1 \vee x_2 \vee x_3 = 1$). This is useful when, for example, a designer knows in advance that some combinations of input bits cannot happen in their specific mode of operation and want to take advantage of this fact. Depending on the complexity of the constraints, it may not be straightforward to otherwise incorporate the constraints without redesigning the logic.

To exploit this feature, we impose constraints in input space as follows. For each function, we generate a random constraint in the input space. The constraint is expressed in conjunctive normal form (CNF), since it is straightforward to be converted into product terms in the DC set. Furthermore, we ask that each clause be the disjunction of at least two literals, since any single-literal clause in the constraint (e.g., $\neg x_i$) can be trivially implemented by assigning a variable to a constant ($x_i = 0$ for the above example) in the original circuit.

An example of the imposed constraint in a 4-input Boolean function $f(x_1, x_2, x_3, x_4)$ is

$$(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3 \vee \neg x_4) = 1,$$

which, according to De Morgan's laws, can be converted into the following DC set in PLA format.

```
00-- -
-101 -
```

where the first four characters in each line shows an input combination that cannot happen, and the rightmost `-` indicates the output is DC.

We use an exponentially delaying probability distribution to control the complexity of the constraint while guaranteeing any constraint is possible to occur in our experiments. Specifically, the constraint has c clauses with probability $1/2^c$ ($c \geq 1$). In clause i, the number of literals is v_i with probability $1/2^{v_i-1}$ ($v_i \geq 2$). In the above example, $c = 2$, $v_1 = 2$, $v_2 = 3$. On average, such a constraint contains 2 clauses with 3 literals in each clause, and invalidates $2/7$ of the original input space.

The constraint in input space is totally optional, without which our proposed framework still works with no compromise in performance.

7.5 Experimental Results

We use 15 random logic functions from the IWLS 2020 benchmark suite [45] which are part of MCNC benchmarks. (Other functions in IWLS 2020 benchmark suite are either special/arithmetic functions—which are not the focus of this chapter, or those of which the truth table is undefined (ML/computer vision problems), or from a commercial design (PicoJava I) that is not publicly available.) Since the IWLS 2020 benchmark contains only a very small part of truth table for each function, we extract the full truth table with `abc` from the MCNC source [70] so that we can simulate any input vector. Each function is the extracted *logic cone* of a primary output of a circuit in the benchmark. (See [45] for details.) For each function we report the number of primary inputs (of the logic cone) in column 1 of Table II. All experiments are performed on a computer with Linux OS, 3.60 GHz CPU, and 64 GB memory. All programs are implemented in C++. All DTs are implemented with XGBoost library [14] with parameters `n_estimators=1`, `max_depth=1000`, `tree_method='exact'` and `random_state=0`. Unless otherwise specified, each model is set with the above parameters. Other XGBoost parameters are set to default.

We compare the following approximate models in our experiments:

- Unregularized DT: This is a DT implementation where we additionally set `min_child_weight=0` and `lambda=0` to disable the default regularization in XGBoost, so that the tree grows until all leaf nodes are pure, i.e., input vectors in the same leaf node have the same output.

- Regularized DT: This is a DT implementation with regularization by imposing penalties on tree complexity (the default behavior in XGBoost library). The default values `min_child_weight=1` and `lambda=1` are set to regularize the tree.

- ADT (impurity): This is an ADT implementation with conventional impurity-based splitting criterion. We set `max_depth=1` to save runtime, as each split is determined only by the first split from a temporary tree trained by XGBoost.

- ADT (MC SHAP): This is our proposed ADT model guided by the MC estimation of SHAP importance. This model does not rely on XGBoost.

- ADT (Tree SHAP): This is our proposed ADT model based on tree estimation of SHAP importance. For each split, a temporary tree is trained by XGBoost specifically for estimating the SHAP importance, and is discarded thereafter.

For simplicity, we refer to the last three ADT models as "impurity model," "MC SHAP model" and "Tree SHAP model," respectively.

Following the logic synthesis flow in Figure 7.2(c), we pass the truth table into the ADT module, feed the resulting approximated PLA to `espresso`, apply DC terms as described in Section 7.4.4, and run `abc` with `read_pla; fraig; ps; write_aiger` to get the reported area and the synthesized AIG. The `abc` synthesis flow is performed for two PLAs before and after adding constraints; the AIG with smaller area will be taken as final.

7.5.1 Error, Area and Runtime of Different Models

We present the experimental results in Table 7.2. For each function, we first convert the exact version (i.e., 100% accurate) of the function into PLA, and feed it to the synthesis flow (i.e., `espresso` followed by `abc`) to create a baseline AIG[4]. This is denoted in the table by "exact full" because the *exact* synthesis is performed over the *full* input space. Next we constrain the input space using the process described in Section 7.4.4, and run the same synthesis flow again. This variation is denoted by "exact constrained" in the table. Column 3 in the table reports the fraction of valid input space compared to the full after constraining it. We report the area of synthesis with exact constrained method relative to the exact full as the ratio of their PLA gate counts in column 5. The error of the exact constrained case is 0 as reported in column 4 because no approximation is used and only the input space is constrained. As shown in column 5, constraining the input space results in a relative area of 0.965, or only 3.5% smaller on average compared to not constraining it, when no approximation is used.

Next, with the same constraints applied in the input space, we run each approximate model with varying parameters that control the number of samples used for approximation/training, followed by the same synthesis flow. For ADT models, the threshold of the number of free bits for sample enumeration is set to 16. (This parameter is not applicable to other models.) We set up an error bound of 0.5%, and report the error rate (in percentage), relative area (see below), and runtime (in seconds) for each function in the following manner. For each model, we report the variation of the model with the parameter that yields the smallest area and an error within the error bound. In case of tie in area, the parameter with the smallest error

[4]Our approximation techniques are compatible with and orthogonal to other logic synthesis and optimization techniques (e.g., `resyn` in `abc`). Once an approximated AIG has been created, any other logic synthesis and optimization techniques may be used to further optimize it. For this reason, we use the AIG before optimization as the baseline in this work.

is preferred. If no parameter yields an error within the error bound, we report with the parameter that yields the smallest error. Similar to the exact constrained case, we report the area *relative* to the exact full case as the ratio of the gate count from each model (as reported by `abc`) to the gate count of the baseline ($\mathrm{GC}_{\mathrm{PLA}}$ given in column 1) and is thus unitless. In other words, an area of 1 means the resulting AIG has the same gate count as in the baseline. The error rate is evaluated with 640K random samples from the constrained input space. The runtimes include all components in the flow, performed with a single CPU core. For the approximate models, all experiments are repeated 10 times with different random seeds to generate the constraints and the results are averaged.

Table 7.2 shows that Unregularized DT, although with a small average error, yields a larger area than the baseline for 12 out of 15 functions, and is $85\times$ larger than the baseline on average. Regularized DT results in smaller areas than Unregularized DT for most functions. However, the areas are still much larger than the baseline on average. This behavior is due to overfitting in the deeper part of the tree, where the number of samples in a node can be very small (1 in the worst case), which is no longer representative of the subfunction. As a result, the learned subfunctions tend to be random at each leaf node that is deep in the tree, the hence overall function becomes very complex. Therefore, they are not suitable for ALS where area is the main focus.

The three ADT models yield much smaller areas than Regularized DT in general. Among the three ADT models, ADT with impurity achieves 28% area reduction with 0.21% error rate. The two ADT models that adopt SHAP importance as the splitting criterion achieve even better area and runtime. They are effective to reduce 39%–42% of the original area on average with an average error rate of 0.20%–0.22%. Specifically, MC SHAP model reduces the area by 39% with 0.22% output error rate on average; Tree SHAP model reduces the area by 42% with 0.20% output error rate.

Table 7.2: Approximation errors, relative area (unitless) and runtimes (in seconds) with different models and an error bound of 0.5%

Function	$\text{Exact}_{\text{full}}$	Input	$\text{Exact}_{\text{constrained}}$			Unregularized DT			Regularized DT			ADT (impurity)			ADT (MC SHAP)			ADT (Tree SHAP)		
(#inputs)	GC_{PLA}	space%	Err%	Area	Time	Err%	Area	Time	Err%	Area	Time	Err%	Area	Time	Err%	Area	Time	Err%	Area	Time
ex60 (43)	296	71.3	0	0.970	<1	0.279	7.23	18	0.442	1.99	16	0.326	0.627	17	0.482	0.598	8	0.324	0.525	23
ex61 (37)	40	72.0	0	1.000	<1	0.276	1.47	4	0.022	0.33	6	0.485	0.173	2	0.485	0.173	3	0.485	0.173	2
ex62 (52)	875	74.2	0	0.997	2	0.048	6.22	24	0.366	1.35	23	0.338	0.753	24	0.364	0.451	18	0.358	0.608	30
ex63 (38)	136	67.9	0	0.982	<1	0.445	1.13	6	0.427	0.41	7	0.075	0.296	4	0.151	0.289	4	0.076	0.296	4
ex64 (47)	147	73.9	0	0.808	<1	0.001	0.05	7	0.001	0.05	7	0.001	0.048	4	0.394	0.034	4	0.001	0.048	4
ex65 (19)	35	71.0	0	0.903	<1	0.000	1.48	5	0.079	0.37	4	0.079	0.371	2	0.455	0.511	2	0.079	0.371	2
ex66 (47)	635	73.9	0	0.949	<1	0.275	4.17	33	0.423	1.47	22	0.243	0.528	31	0.161	0.557	16	0.345	0.572	37
ex67 (46)	769	75.7	0	0.921	1	0.259	3.41	28	0.398	1.16	23	0.191	0.546	27	0.368	0.336	10	0.203	0.520	37
ex68 (33)	97	67.8	0	0.987	86	0.249	307.95	36	1.024	90.36	30	0.588	1.915	1594	0.233	0.808	708	0.134	0.886	941
ex69 (16)	30	83.0	0	0.973	<1	0.000	0.97	5	0.427	3.45	3	0.000	0.973	1	0.000	0.973	1	0.000	0.973	1
ex70 (23)	60	65.3	0	0.995	<1	0.277	169.50	33	0.645	0.11	4	0.348	0.868	4	0.000	1.013	9	0.400	0.890	5
ex71 (23)	51	65.3	0	0.994	<1	0.052	417.21	37	0.812	108.62	11	0.416	0.994	6	0.000	0.994	9	0.460	0.182	13
ex72 (35)	96	76.6	0	1.000	<1	0.318	10.84	11	0.394	4.98	8	0.095	0.593	7	0.217	0.478	8	0.095	0.593	9
ex73 (16)	94	83.0	0	1.000	<1	0.000	1.00	130	0.000	1.00	158	0.000	1.000	66	0.000	1.000	65	0.000	1.000	68
ex74 (16)	45	83.0	0	0.993	16	0.259	340.56	88	3.431	1060.2	196	0.000	0.993	37	0.000	0.993	55	0.000	0.993	39
Mean	227	73.6	0	0.965	7	0.182	84.88	31	0.593	85.06	35	0.212	0.712	122	0.221	0.614	61	0.199	0.575	81

All three ADT models consume a runtime up to minutes for each function, and MC SHAP consumes generally shorter runtime than the other two.

Note that ex68 consumes much longer runtime than others due to its function complexity, especially the large number of XOR functions for this primary output. As a result, much more product terms are generated in the resulting PLA before minimization. This prolongs both the ADT development and logic minimization.

To better observe the tradeoffs among error, area and runtime, we vary the parameters in three ADT models with different splitting criteria, and plot the curves of error vs area, and error vs runtime in Figure 7.4. In both plots, a curve closer to the left-bottom corner has a better tradeoff. Unregularized DT and Regularized DT models are excluded due to much larger area.

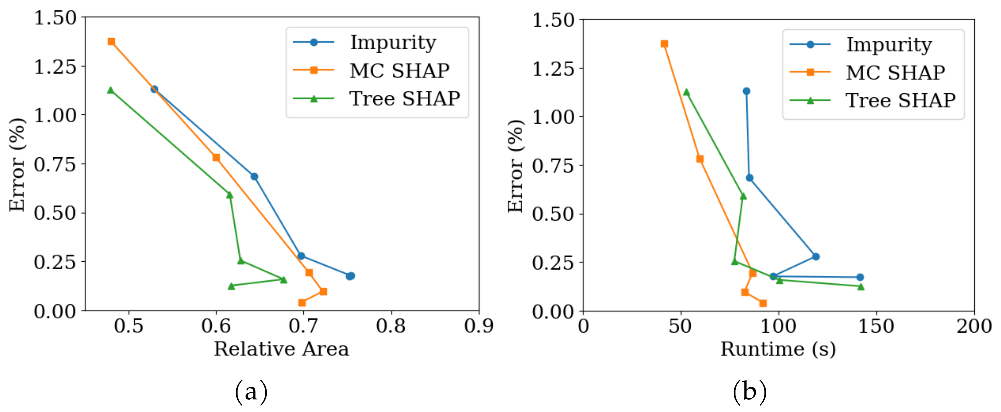


Figure 7.4: Comparison of tradeoffs (a) between error and area, and (b) between error and runtime for ADT models with different splitting criteria: impurity, MC SHAP, and Tree SHAP. Values are averaged over all functions. Points on each curve are generated by varying r in {128, 256, 512, 1024, 2048}.

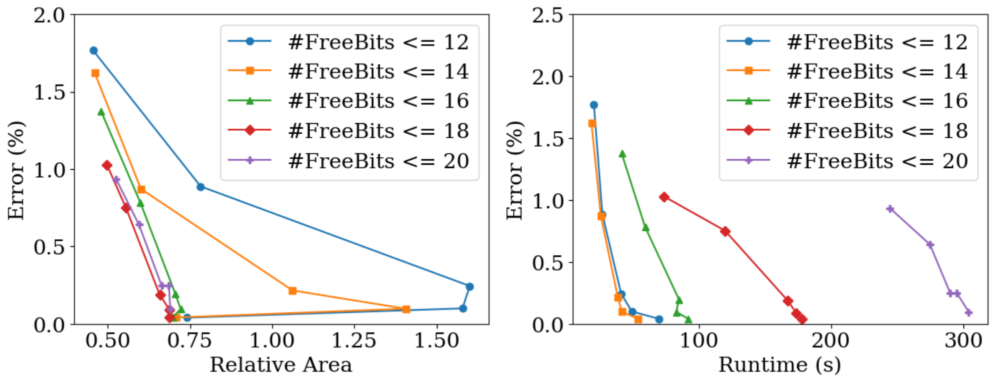
Figure 7.4(a) shows that Tree SHAP has the best tradeoff among the three models. ADT with impurity has the worse trade-off compared to other two models that are guided by SHAP importance. This observation means that in the context of ALS, SHAP importance is a better metric to determine the splits than the impurity-based metric used in ADT. Tree SHAP model has a superior tradeoff curve to MC SHAP model, owing to the more accurate estimation of SHAP values and the default behavior of tree regularization in XGBoost library. However, MC SHAP can reach smaller error than Tree SHAP when more area budget is allowed, where the error of other two models may not go to zero due to tree regularization. Therefore, it is suggested to prioritize the usage of Tree SHAP model when the error budget is higher ($> 0.2\%$) and MC SHAP otherwise.

Figure 7.4(b) shows that ADT models with impurity and Tree SHAP can take longer time to run if a small error is desired. And ADT with impurity is the least runtime-efficient model. In contrast, the MC SHAP model consumes shorter runtime even with small errors.

7.5.2 Impact of threshold for sample enumeration in ADTs

We further explore the impact of threshold of $nFreeBits$ in MC SHAP model, which controls when to stop splitting the tree and begin enumerating samples, as detailed in Section 7.4.2. The results are only shown for MC SHAP but a similar trend is observed for other ADT methods (Tree SHAP and ADT with impurity). We run the experiments with thresholds of 12, 14, 16, 18 and 20, respectively, and plot the tradeoffs among error, area, and runtime with different thresholds in Figure 7.5.

From Figure 7.5(a), we can observe similar tradeoff curves for thresholds of 16, 18 and 20, where the thresholds of 18 and 20 yield slightly better tradeoffs. Thresholds of 12 and 14 yield suboptimal results due to overfitting for a few functions, observed as the excessive area in some error range. In addition, Figure 7.5(b) shows that thresholds that are too large (18 and 20) can lead to suboptimal tradeoff between error and runtime, especially for complex functions that consume longer runtime (e.g., ex68).



(a) (b)

Figure 7.5: Comparison of tradeoffs (a) between error and area, and (b) between error and runtime, with MC SHAP and different thresholds for the number of free bits in a node to trigger sample enumeration. Error, area and runtime values are averaged over all functions. Points on each curve are generated by varying r in $\{128, 256, 512, 1024, 2048\}$.

Therefore, a threshold around 16 is recommended, which is aligned with our discussion in Section 7.4.2.

7.6 Conclusion

In this chapter, we proposed adaptive decision trees (ADTs) with emphasis on variations guided by SHAP explainability from ML as a promising technique for sampling-based approximate logic synthesis. Specific approximation techniques for ADT were proposed to achieve an efficient and effective implementation including approximation of don't-care and XOR bits. Comprehensive experiments showed the effectiveness of SHAP-guided ADT in significantly reducing the area with a small tradeoff with accuracy.

8 CONCLUSION AND FUTURE DIRECTIONS

In this dissertation, we presented three techniques where SHAP—a recent development in XAI—was incorporated in different aspects and stages of VLSI design, including logic synthesis, physical design, and hardware security. All of these XAI-powered techniques achieved better results in efficiency and effectiveness than conventional techniques or existing works in similar settings, being ML-based or not. These techniques set examples of how XAI, or more specifically, explainable ML can be incorporated into VLSI design to achieve better design outcomes. With the same idea, some techniques presented in this dissertation can be extended. For example, in Chapter 5, we obfuscated the layout by altering the routing. However, as the explanatory analysis can be generalized for any ML attack, its usage is not limited to routing obfuscation. It is possible to incorporate it into other defense techniques (e.g. placement obfuscation) and/or a mix of multiple techniques in the future.

This dissertation mainly focused on the usage of SHAP—a specific tool that is actively being developed in XAI community. Although not demonstrated in this dissertation, SHAP is also compatible with some deep NN architectures, such as feed-forward NNs, CNNs, and Recurrent NNs. However, with current development, SHAP is still computationally expensive, except for tree-based ML models, with special assumptions, or with compromised accuracy. As complex models like CNN become increasingly popular in various applications including VLSI design techniques, it would be very interesting and demanding to explore computationally-efficient ways to utilize the explainability of these models.

In addition, there is a gap between the development of ML algorithms and the XAI counterparts. For example, SHAP is not yet compatible with the paradigm of reinforcement learning, which has been applied in logic synthesis, optimization, and physical design. Supporting for newer NN architecture (e.g., Graph NN) is also yet to be developed, whereas GNN already has applications in testability, power, and signal integrity analysis, and is promising on logic synthesis and optimization owing to the natural similarity between graphs and netlists. Therefore, it is highly desirable for researchers to develop explanatory analysis techniques that support these new models, so that more ML-based techniques in VLSI designs may benefit from the added explainability of corresponding ML models.

Last but not least, there is a need for verification of the explanations generated by XAI techniques. In the field of VLSI design, it may require participation of both the academia and the industry, due to the extremely high complexity of real-world circuits and therefore the heavy development efforts on electronic design automation (EDA) tools. For example, for explainable routability prediction in Chapter 6, support from EDA tool vendors will be needed to incorporate the explanations for DRC violation hotspots and the corresponding actions into the EDA tools, in order to comprehensively verify the accuracy and effectiveness of such explanations in real practice.

[1] M. B. Alawieh, W. Ye, and D. Z. Pan, "Re-examining VLSI manufacturing and yield through the lens of deep learning," in *Proceedings of the 2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, 8 pages.

[2] V. Arya, R. K. E. Bellamy, P.-Y. Chen, A. Dhurandhar, M. Hind, S. C. Hoffman *et al.*, "One explanation does not fit all: A toolkit and taxonomy of AI explainability techniques," 2019, arXiv:1909.03012.

[3] J. Baehr, A. Bernardini, G. Sigl, and U. Schlichtmann, "Machine learning and structural characteristics for reverse engineering," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference (ASPDAC)*, 2019, pp. 96–103.

[4] S. Boroumand, C.-S. Bouganis, and G. A. Constantinides, "Learning Boolean circuits from examples for approximate logic synthesis," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference (ASPDAC)*, 2021, pp. 524–529.

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

[6] I. S. Bustany, D. Chinnery, J. R. Shinnerl, and V. Yutsis, "ISPD 2015 benchmarks with fence regions and routing blockages for detailed-routing-driven placement," in *Proceedings of the 2015 ACM International Symposium on Physical Design (ISPD)*, 2015, pp. 157–164.

[7] P. Chakraborty, J. Cruz, and S. Bhunia, "SAIL: Machine learning guided structural analysis attack on hardware obfuscation," in *Proceedings of the 2018 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, 2018, pp. 56–61.

[8] W.-T. J. Chan, Y. Du, A. B. Kahng, S. Nath, and K. Samadi, "BEOL stack-aware routability prediction from placement using data mining techniques," in *Proceedings of the 2016 IEEE 34th International Conference on Computer Design (ICCD)*, 2016, pp. 41–48.

[9] W.-T. J. Chan, P.-H. Ho, A. B. Kahng, and P. Saxena, "Routability optimization for industrial designs at sub-14nm process nodes using machine learning," in *Proceedings of the 2017 ACM International Symposium on Physical Design (ISPD)*, 2017, pp. 15–21.

[10] S. Chatterjee, "Learning and memorization," in *Proceedings of the 35th International Conference on Machine Learning (ICML)*, vol. 80, 2018, pp. 755–763.

[11] H. Chen and D. S. Boning, "Machine learning approaches for IC manufacturing yield enhancement," in *Machine Learning in VLSI Computer-Aided Design*, I. A. M. Elfadel, D. S. Boning, and X. Li, Eds. Springer International Publishing, 2019, pp. 175–199.

[12] L.-C. Chen, C.-C. Huang, Y.-L. Chang, and H.-M. Chen, "A learning-based methodology for routability prediction in placement," in *Proceedings of the 2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, 2018, 4 pages.

[13] P.-W. Chen, Y.-C. Huang, C.-L. Lee, and J.-H. R. Jiang, "Circuit learning for logic regression on high dimensional Boolean space," in *Proceedings of the 57th Design Automation Conference (DAC)*, 2020, Art. no. 15.

[14] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2016, pp. 785–794.

[15] V. A. Chhabria, V. Ahuja, A. Prabhu, N. Patil, P. Jain, and S. S. Sapatnekar, "Thermal and IR drop analysis using convolutional encoder-decoder networks," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference (ASPDAC)*, 2021, pp. 690–696.

[16] C.-C. Chi and J.-H. R. Jiang, "Logic synthesis of binarized neural networks for efficient circuit implementation," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 2018, Art. no. 84.

[17] N. K. Darav, A. Kennings, A. F. Tabrizi, D. Westwick, and L. Behjat, "Eh?placer: a high-performance modern technology-driven placer," *ACM Transactions on Design Automation of Electronic Systems*, vol. 21, no. 3, July 2016, Art. no. 37.

[18] S. Dash, O. Gunluk, and D. Wei, "Boolean decision rules via column generation," in *Advances in Neural Information Processing Systems 31 (NeurIPS)*, 2018, pp. 4655–4665.

[19] J. Davis and M. Goadrich, "The relationship between precision-recall and roc curves," in *Proceedings of the 23rd International Conference on Machine learning (ICML)*, 2006, pp. 233–240.

[20] B. A. de Abreu, A. Berndt, I. S. Campos, C. Meinhardt, J. T. Carvalho, M. Grellert *et al.*, "Fast logic optimization using decision trees," in *Proceedings of International Symposium on Circuits and Systems (ISCAS)*, 2021, 5 pages.

[21] G. De'ath and K. E. Fabricius, "Classification and regression trees: a powerful yet simple technique for ecological data analysis," *Ecology*, vol. 81, no. 11, pp. 3178–3192, November 2000.

[22] R. Elnaggar and K. Chakrabarty, "Machine learning for hardware security: Opportunities and risks," *Journal of Electronic Testing*, vol. 34, no. 2, pp. 183–201, April 2018.

[23] E. Frank, M. A. Hall, and I. H. Witten, *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"*, 4th ed. Morgan Kaufmann, 2016.

[24] S. Gogri, J. Hu, A. Tyagi, M. Quinn, S. Ramachandran, F. Batool *et al.*, "Machine learning-guided stimulus generation for functional verification," in *Proceedings of the 2020 Design and Verification Conference (DVCon)*, 2020, 10 pages.

[25] W. Haaswijk, E. Collins, B. Seguin, M. Soeken, F. Kaplan, S. Süsstrunk *et al.*, "Deep learning for logic optimization algorithms," in *Proceedings of the 2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, 4 pages.

[26] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, July 1968.

[27] M. Hind, D. Wei, M. Campbell, N. C. F. Codella, A. Dhurandhar, A. Mojsilović *et al.*, "TED: Teaching AI to explain its decisions," in *Proceedings of the 2019 AAAI/ACM Conference on AI, Ethics, and Society (AIES)*, 2019, pp. 123–129.

[28] G. Huang, J. Hu, Y. He, J. Liu, M. Ma, Z. Shen *et al.*, "Machine learning for electronic design automation: A survey," *ACM Transactions on Design Automation of Electronic Systems*, vol. 26, no. 5, June 2021, Art. no. 40.

[29] K. Huang, J. M. Carulli, and Y. Makris, "Parametric counterfeit ic detection via support vector machines," in *Proceedings of the 2012 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2012, pp. 7–12.

[30] H. Lin and P. Li, "Classifying circuit performance using active-learning guided support vector machines," in *Proceedings of the 2012 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2012, pp. 187–194.

[31] Y. Lin, M. B. Alawieh, W. Ye, and D. Z. Pan, "Machine learning for yield learning and optimization," in *Proceedings of the 2018 IEEE International Test Conference (ITC)*, 2018, 10 pages.

[32] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," in *Proceedings of the 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013, 7 pages.

[33] S. M. Lundberg, G. Erion, H. Chen, A. DeGrave, J. M. Prutkin, B. Nair *et al.*, "From local explanations to global understanding with explainable AI for trees," *Nature Machine Intelligence*, vol. 2, no. 1, pp. 56–67, January 2020.

[34] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems (NIPS) 30*, 2017, pp. 4765–4774.

[35] Y. Ma, H. Ren, B. Khailany, H. Sikka, L. Luo, K. Natarajan *et al.*, "High performance graph convolutional networks with applications in testability analysis," in *Proceedings of the 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, 6 pages.

[36] J. Magaña, D. Shi, J. Melchert, and A. Davoodi, "Are proximity attacks a threat to the security of split manufacturing of integrated circuits?" *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 12, pp. 3406–3419, December 2017.

[37] A. Mirhoseini, A. Goldie, M. Yazgan, J. Jiang, E. Songhori, S. Wang *et al.*, "Chip placement with deep reinforcement learning," 2020, arXiv:2004.10746.

[38] C. Molnar, *Interpretable Machine Learning*. Lulu Press, 2020. [Online]. Available: https://christophm.github.io/interpretable-ml-book/

[39] R. Netto, S. Fabre, T. A. Fontana, V. Livramento, L. L. Pilla, L. Behjat *et al.*, "Algorithm selection framework for legalization using deep convolutional neural networks and transfer learning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021, 14 pages, to be published.

[40] A. L. Oliveira and A. Sangiovanni-Vincentelli, "Learning complex Boolean functions: Algorithms and applications," in *Advances in Neural Information Processing Systems 6 (NIPS)*, 1993, pp. 911–918.

[41] G. Pasandi, S. Nazarian, and M. Pedram, "Approximate logic synthesis: A reinforcement learning-based technology mapping approach," in *Proceedings of the 20th International Symposium on Quality Electronic Design (ISQED)*, 2019, pp. 26–32.

[42] S. Patnaik, M. Ashraf, J. Knechtel, and O. Sinanoglu, "Raise your game for split manufacturing: Restoring the true functionality through BEOL," in *Proceedings of the 55th Annual Design Automation Conference (DAC)*, 2018, Art. no. 140.

[43] S. Patnaik, J. Knechtel, M. Ashraf, and O. Sinanoglu, "Concerted wire lifting: Enabling secure and cost-effective split manufacturing," in *Proceedings of the 23rd Asia and South Pacific Design Automation Conference (ASPDAC)*, 2018, pp. 251–258.

[44] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, March 1986.

[45] S. Rai, W. L. Neto, Y. Miyasaka, X. Zhang, M. Yu, Q. Yi *et al.*, "Logic synthesis meets machine learning: Trading exactness for generalization," in *Proceedings of Design, Automation Test in Europe Conference and Exhibition (DATE)*, 2021, pp. 1026–1031.

[46] J. Rajendran, O. Sinanoglu, and R. Karri, "Is split manufacturing secure?" in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2013, pp. 1259–1264.

[47] G. R. Reddy, K. Madkour, and Y. Makris, "Machine learning-based hotspot detection: Fallacies, pitfalls and marching orders," in *Proceedings of the 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, 8 pages.

[48] M. T. Ribeiro, S. Singh, and C. Guestrin, "'Why should I trust you?': Explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2016, pp. 1135–1144.

[49] L. Rokach, M. Kalech, G. Provan, and A. Feldman, "Machine-learning-based circuit synthesis," in *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI)*, 2013, pp. 1635–1641.

[50] M. Sandri and P. Zuccolotto, "Analysis and correction of bias in total decrease in node impurity measures for tree-based algorithms," *Statistics and Computing*, vol. 20, no. 4, pp. 393–407, October 2010.

[51] I. Scarabottolo, G. Ansaloni, G. A. Constantinides, L. Pozzi, and S. Reda, "Approximate logic synthesis: A survey," *Proceedings of the IEEE*, vol. 108, no. 12, pp. 2195–2213, December 2020.

[52] U. Schlichtmann, S. Das, I. Lin, and M. P. Lin, "Overview of 2019 CAD contest at ICCAD," in *Proceedings of the 2019 IEEE/ACM International Conference on Computer-Aided Design* (*ICCAD*), 2019, 2 pages.

[53] K. Shamsi, D. Z. Pan, and Y. Jin, "On the impossibility of approximation-resilient circuit locking," in *Proceedings of the 2019 IEEE International Symposium on Hardware Oriented Security and Trust* (*HOST*), 2019, pp. 161–170.

[54] A. F. Tabrizi, N. K. Darav, L. Rakai, A. Kennings, and L. Behjat, "Detailed routing violation prediction during placement using machine learning," in *Proceedings of the 2017 International Symposium on VLSI Design, Automation and Test* (*VLSI-DAT*), 2017, 4 pages.

[55] A. F. Tabrizi, N. K. Darav, S. Xu, L. Rakai, I. Bustany, A. Kennings *et al.*, "A machine learning framework to identify detailed routing short violations from a placed netlist," in *Proceedings of the 55th Annual Design Automation Conference* (*DAC*), 2018, Art. no. 48.

[56] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "SALSA: Systematic logic synthesis of approximate circuits," in *Proceedings of the 49th Annual Design Automation Conference* (*DAC*), 2012, pp. 796–801.

[57] C.-K. Wang, C.-C. Huang, S. S.-Y. Liu, C.-Y. Chin, S.-T. Hu, W.-C. Wu *et al.*, "Closing the gap between global and detailed placement: Techniques for improving routability," in *Proceedings of the 2015 ACM International Symposium on Physical Design* (*ISPD*), 2015, pp. 149–156.

[58] Y. Wang, T. Cao, J. Hu, and J. Rajendran, "Front-end-of-line attacks in split manufacturing," in *Proceedings of the 36th IEEE/ACM International Conference on Computer-Aided Design* (*ICCAD*), 2017, 8 pages.

[59] Y. Wang, P. Chen, J. Hu, G. Li, and J. Rajendran, "The cat and mouse in split manufacturing," *IEEE Transactions on Very Large Scale Integration* (*VLSI*) *Systems*, vol. 26, no. 5, pp. 805–817, May 2018.

[60] Y. Wang, P. Chen, J. Hu, and J. Rajendran, "Routing perturbation for enhanced security in split manufacturing," in *Proceedings of the 2017 22nd Asia and South Pacific Design Automation Conference (ASPDAC)*, 2017, pp. 605–610.

[61] Y. Wei, C. Sze, N. Viswanathan, Z. Li, C. J. Alpert, L. Reddy *et al.*, "GLARE: Global and local wiring aware routability evaluation," in *Proceedings of the 49th Annual Design Automation Conference (DAC)*, 2012, pp. 768–773.

[62] K. Xiao, D. Forte, and M. M. Tehranipoor, "Efficient and secure split manufacturing via obfuscated built-in self-authentication," in *Proceedings of the 2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2015, pp. 14–19.

[63] Z. Xie, G.-Q. Fang, Y.-H. Huang, H. Ren, Y. Zhang, B. Khailany *et al.*, "FIST: A feature-importance sampling and tree-based method for automatic design flow parameter tuning," in *Proceedings of the 2020 25th Asia and South Pacific Design Automation Conference (ASPDAC)*, 2020, pp. 19–25.

[64] Z. Xie, Y. Huang, G. Fang, H. Ren, S. Fang, Y. Chen *et al.*, "RouteNet: routability prediction for mixed-size designs using convolutional neural network," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 2018, Art. no. 80.

[65] J. Xiong, Y. Zhu, and J. He, "Machine learning for VLSI chip testing and semiconductor manufacturing process monitoring and improvement," in *Machine Learning in VLSI Computer-Aided Design*, I. A. M. Elfadel, D. S. Boning, and X. Li, Eds. Springer International Publishing, 2019, pp. 233–263.

[66] W. Xu, L. Feng, J. Rajendran, and J. Hu, "Layout recognition attacks on split manufacturing," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference (ASPDAC)*, 2019, pp. 45–50.

[67] H. Yang, Y. Lin, B. Yu, and E. F. Y. Young, "Lithography hotspot detection: From shallow to deep learning," in *Proceedings of the 30th IEEE International System-on-Chip Conference (SOCC)*, 2017, pp. 233–238.

[68] H. Yang, W. Zhong, Y. Ma, H. Geng, R. Chen, W. Chen *et al.*, "VLSI mask optimization: From shallow to deep learning," in *Proceedings of the 25th Asia and South Pacific Design Automation Conference (ASPDAC)*, 2020, pp. 434–439.

[69] H. Yang, S. Li, Y. Ma, B. Yu, and E. F. Y. Young, "GAN-OPC: Mask optimization with lithography-guided generative adversarial nets," in *Proceedings of the 55th Annual Design Automation Conference (DAC)*, 2018, Art. no. 131.

[70] S. Yang, *Logic Synthesis and Optimization Benchmarks User Guide: Version 3.0.* Microelectronics Center of North Carolina (MCNC), 1991.

[71] W. Ye, M. B. Alawieh, Y. Lin, and D. Z. Pan, "LithoGAN: End-to-end lithography modeling with generative adversarial networks," in *Proceedings of the 56th Annual Design Automation Conference (DAC)*, 2019, Art. no. 107.

[72] W. Zeng, B. Zhang, and A. Davoodi, "Analysis of security of split manufacturing using machine learning," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 12, pp. 2767–2780, December 2019.

[73] W. Zeng, A. Davoodi, and R. O. Topaloglu, "Explainable DRC hotspot prediction with random forest and SHAP tree explainer," in *Proceedings of the 2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 1151–1156, doi: 10.23919/DATE48585.2020.9116488.

[74] ——, "ObfusX: Routing obfuscation with explanatory analysis of a machine learning attack," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference (ASPDAC)*, 2021, pp. 548–554, doi: 10.1145/3394885.3431600.

[75] ——, "Sampling-based approximate logic synthesis: An explainable machine learning approach," in *Proceedings of the 2021 IEEE/ACM International Conference on Computr-Aided Design (ICCAD)*, 2021, to be published.

[76] Y. Zhang, H. Ren, and B. Khailany, "GRANNITE: Graph neural network inference for transferable power estimation," in *Proceedings of the 57th Design Automation Conference (DAC)*, 2020, Art. no. 60, 6 pages.

[77] Y. Zhang and Y. Makris, "Hardware-based detection of spectre attacks: A machine learning approach," in *Proceedings of the 2020 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, 2020, 6 pages.

[78] Q. Zhou, X. Wang, Z. Qi, Z. Chen, Q. Zhou, and Y. Cai, "An accurate detailed routing routability prediction model in placement," in *Proceedings of the 6th Asia Symposium on Quality Electronic Design (ASQED)*, 2015, pp. 119–122.

[79] K. Zhu, M. Liu, H. Chen, Z. Zhao, and D. Z. Pan, "Exploring logic optimizations with reinforcement learning and graph convolutional network," in *Proceedings of the 2020 ACM/IEEE 2nd Workshop on Machine Learning for CAD (MLCAD)*, 2020, pp. 145–150.