Machine Learning Based Routing Congestion Prediction in FPGA High-Level Synthesis

Jieru Zhao*, Tingyuan Liang*, Sharad Sinha† and Wei Zhang*
*Department of ECE, Hong Kong University of Science and Technology, {jzhaoao,tliang,wei.zhang}@ust.hk
†Department of CSE, Indian Institute of Technology Goa, sharad_sinha@ieee.org

***Abstract*—High-level synthesis (HLS) shortens the development time of hardware designs and enables faster design space exploration at a higher abstraction level. Optimization of complex applications in HLS is challenging due to the effects of implementation issues such as routing congestion. Routing congestion estimation is absent or inaccurate in existing HLS design methods and tools. Early and accurate congestion estimation is of great benefit to guide the optimization in HLS and improve the efficiency of implementation. However, routability, a serious concern in FPGA designs, has been difficult to evaluate in HLS without analyzing post-implementation details after Place and Route. To this end, we propose a novel method to predict routing congestion in HLS using machine learning and map the expected congested regions in the design to the relevant high-level source code. This is greatly beneficial in early identification of routability oriented bottlenecks in the high-level source code without running time-consuming register-transfer level (RTL) implementation flow. Experiments demonstrate that our approach accurately estimates vertical and horizontal routing congestion with errors of 6.71% and 10.05% respectively. By presenting *Face Detection* application as a case study, we show that by discovering the bottlenecks in high-level source code, routing congestion can be easily and quickly resolved compared to the efforts involved in RTL level implementation and design feedback.**

I. Introduction

Field-programmable gate arrays (FPGAs) can be reprogrammed to implement various tasks with low power and massive parallelism. Due to increasing demands of FPGA applications, high-level synthesis (HLS) becomes a superior alternative to register-transfer level (RTL) methods for its higher productivity. By generating RTL from behavioral descriptions, HLS tools provide synthesis directives and enable fast design space exploration to meet design requirements [1]. However, it is much harder to grasp the hardware implementation details when designing at a higher level of abstraction. Although HLS tools provide estimations of several post-implementation metrics, the estimated values deviate significantly from the actual values because the subsequent optimization in the downstream implementation steps is not modeled [2, 3]. This inaccuracy affects the correct evaluation of the design quality, misleads designers during the performance optimization process and incurs implementation issues that degrade the final performance. One of the crucial issues is routing congestion.

Routing is an important problem in FPGA designs since it contributes a lot to delay and resource utilization. In a congested design, wires have to be detoured for connections, generating longer delays and occupying more routing resources [4]. Routing congestion degrades the design performance and even leads to implementation failures. Therefore, it is vital that routing congestion could be identified and resolved early in the design cycle. Detection of routing congestion at the HLS source-code level could reduce the iterations of the design cycle and help designers choose a proper optimization scheme.

In physical design, many works [5]–[7] predict routing congestion to guide FPGA placement. With informative physical metrics, routing congestion can be predicted with high accuracy. However, as the abstraction level increases, the difficulty of congestion estimation is exacerbated due to the lack of physical information. Several HLS-based methods [8]–[11] are proposed to improve HLS scheduling or allocation algorithms to generate layout-friendly RTL models. [8]–[10] incorporate floorplanning to HLS and [11] summarizes multiple RTL metrics to evaluate the quality of HLS-generated models. All of these methods aim at improving HLS algorithms, which is different from our problem that how to detect and eliminate congestion issues in the source code. [12] identifies congestion in HLS by integrating a physically aware logic synthesis tool from Cadence. The required physical information is obtained from the generated congestion reports and no estimation is needed. Since we focus on FPGA-based designs and utilize FPGA commercial design tools like Vivado, related congestion metrics can only be obtained after Place and Route (PAR).

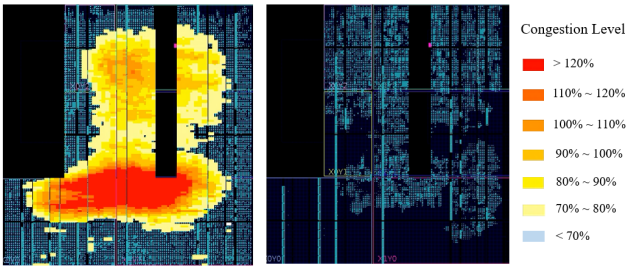
To this end, we propose a machine-learning based method to predict routing congestion in FPGA high-level synthesis and locate the highly congested regions in the source code. Based on the accurate congestion detection in HLS, routing congestion can be resolved easily at the early design stage. To the best of our knowledge, we are the first to build a congestion prediction model in FPGA high-level synthesis to help designers solve the routing congestion issues and optimize their applications. Our major contributions include: (1) To construct the dataset for model training, we develop an effective method to back trace the congestion metrics of CLBs and link with the HLS IR; (2) We propose seven informative categories of features and train multiple machine learning models to investigate the impact of our features. (3) Based on our trained model, the highly congested parts of the source code can be detected and we introduce several methods to reduce congestion in HLS and enhance design performance.

II. Motivation

We take an FPGA-based design *Face Detection* from Rosetta [13] as example to show the routing congestion issue in HLS. To meet the constraints of real-time applications, the designers apply several HLS directives such as function inlining, loop pipelining and unrolling, and array partitioning in [13]. We compare two implementations on a Xilinx Zynq

TABLE I. PERFORMANCE COMPARISON

Implementation	WNS(ns)	Max Freq.(MHz)	Latency(cycles)	Max Congestion(%)
With Directives	-13.643	42.3	1.08e+6	178.96
Without Directives	-0.066	99.3	1.73e+7	58.51



(a) With directives (b) Without directives

Fig. 1: Congestion maps of *Face Detection* from Vivado

device (xc7z020clg484): one is the original implementation applied with the directives mentioned above, and the other one is the implementation without any directive applied. Figure 1 shows the congestion maps from Xilinx Vivado and the color represents the degree of congestion. Congestion level denotes the percentage of routing resources used in corresponding tiles. A value over 100% means that over 100% of routing resources will be used and the router has to divert routes around that area. Table I compares multiple design performance metrics of the two implementations. The worst negative slack (WNS) indicates how much the design is missing the timing constraints.

We can see that implementations of the same design can vary greatly in performance if different optimization methods are adopted in HLS tools. By applying directives, the latency is reduced significantly, but the implementation becomes much more congested and the maximum frequency is decreased. For hardware designs, it is crucial to consider the trade-off among design performance metrics and select the right implementation method to meet the design constraints. However, current HLS tools like Vivado HLS fail to provide accurate estimations of physical metrics, since it is non-trivial to model the accumulated effects of each step in the RTL implementation flow. Although HLS can estimate the accurate clock cycle number, it is difficult to estimate the real timing (ns) and find the direction for further performance optimization of the design. At the same time, it takes too much time to repetitively run the complete C-to-FPGA flow to obtain the real timing in every round of design optimization. For example, it takes nearly seven hours to finish the logic synthesis and PAR for the *Face Detection* application, compared to the significantly less time in HLS flow (several minutes). Therefore, it is of great benefit to provide early accurate prediction of physical information in HLS, such as routing congestion, which can guide the optimization and shorten the design cycle.

For this reason, we propose our machine-learning based method to predict routing congestion during HLS and locate the highly congested regions in the source code. By identifying the problems and resolving congestion earlier, the maximum frequency is increased when then latency is still reduced, achieving a better trade-off to maximize the design performance using HLS tools. Note that the target of prediction in our problem is the routing congestion as shown in Fig. 1.

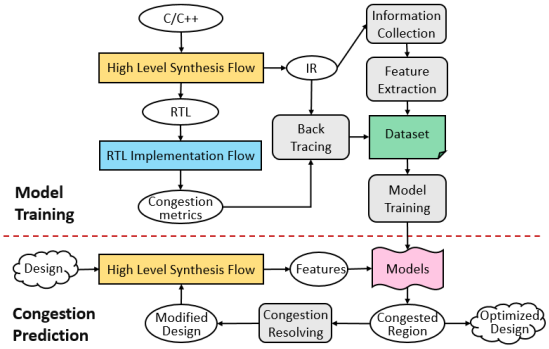


Fig. 2: Overview of the proposed approach.

III. CONGESTION PREDICTION FLOW

Figure 2 shows the overview of our proposed approach. There are two phases: one is to train the routing congestion prediction model and the other one is to predict the highly congested regions in the source code. During the training phase, with several HLS-based applications, we first run one time of the complete C-to-FPGA flow to obtain the routing congestion metrics. The C/C++ specifications of designs are synthesized into RTL models through the HLS flow, and then the RTL descriptions run through the implementation flow to generate the congestion metrics. Our approach begins with the intermediate representation (IR), which is generated by the HLS front-end, and targets at estimating congestion after the RTL implementation. The front-end compiler performs code optimization such as bitwidth reduction, which directly influences the data flow of generated RTL models [14]. By studying at the IR level, source-code transformations are considered and more accurate information can be extracted. To build the dataset for training, we back trace the routing congestion metrics per CLB to the operations in IR and collect necessary information from HLS to extract features. After that, features of each operation are extracted and the dataset is built. Each sample in the dataset consists of features of each operation and labels which are corresponding congestion metrics. With the dataset, we train machine learning models for congestion estimation. If the models are trained with a large dataset built from a variety of applications, routing congestion can be estimated accurately. If there are not many available applications to build a comprehensive dataset, the target design should go through the complete C-to-FPGA flow for one time to generate congestion metrics which will be used to enrich the dataset and improve the estimation accuracy. With the trained model, the highly congested regions in the source code of the target design can be detected during the prediction phase and users can resolve congestion issues in the HLS flow without running the time-consuming RTL implementation flow.

A. Back Tracing and Information Collection

To obtain the dataset, we need to establish the one-to-one relationship between IR operations and congestion metrics and collect required information from the HLS synthesis flow.

1) Back-tracing: HLS synthesizes a design by scheduling IR operations to different control states and binding operations to functional units based on characterized libraries, as shown in

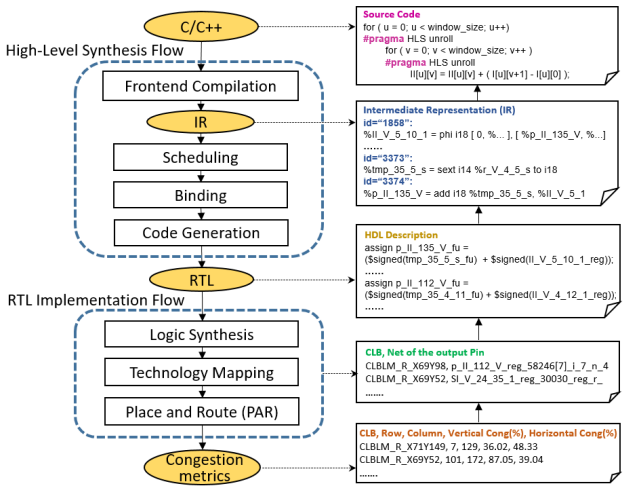


Fig. 3: Back-tracing process.

Fig. 3. It generates the RTL data path and FSM. After that, the RTL implementation flow synthesizes the HDL descriptions into gate-level netlists. The exact location of each operator on FPGA is unknown until PAR. Therefore, automatic back-tracing is employed to find the location of each IR operation on FPGA after PAR and extract corresponding congestion metrics of the CLBs in which the operation is implemented. Based on Xilinx Vivado design suite, our automatic back-tracing flow is shown in Fig. 3, accompanied with code snippets at different steps. Physical information is extracted first, including vertical and horizontal congestion metrics per CLB and coordinates of row and column. For each CLB, the net names of the output pins of each cell are gathered through Vivado, which are then back traced to HDL descriptions. Finally, based on HLS-generated information during the synthesis procedure, we can establish the relationship between RTL operations and IR operations. Note that IR operations can be further back traced to the statements of the source code.

2) Information Collection: To extract features that reflect the quality of hardware implementations like interconnection and resource usage, comprehensive HLS-based information is required to be collected in advance.

Information of each operator. The values of multiple metrics for each operator are obtained from the HLS pre-characterization libraries. We collect the resource usage, operation type, bitwidth and delay (ns) for each operator.

Dependency among operators. To trace the relationship between IR operations for one design, a dependency graph is constructed by storing each operation as one node and connecting dependent operations. The edge weight is measured by counting the number of wires for each connection. Take a 32-bit wide operator as an example, if one of its successors takes eight of the total 32 bits as the input signals, the actual number of wires for this connection is eight which is stored as the edge weight. Resource sharing is considered by replacing the operation nodes which share the same RTL module with one combined node, as shown in Fig. 4. The original nodes are removed and corresponding edges are redirected to the combined node. We also consider the influence of the function interface, which does not correspond to an IR operation but

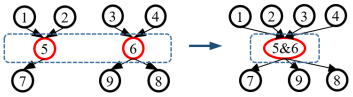


Fig. 4: Merging the nodes that share the same RTL module.

reflects the connections of I/O ports. Therefore, "port" type nodes are also added to the graph to indicate which operators are connected to the same I/O port.

Scheduling and Global information. HLS schedules operations into control states which reflect the overall latency of a design and the latest arrival time of the input signals for each operation. After keeping a record of the control states during which one operation is executed, exact operation latencies are obtained and the maximum distance between dependent operations could be measured. Detailed explanation is given in Section III-B. Global information is extracted based on HLS estimations, such as the resource usage and latencies of the functions and the total number of multiplexers.

B. Feature Extraction

To capture the characteristics of each operation in different designs, we extract 302 related features and divide them into seven categories, as listed in Table II. For better understanding, several categories are explained in detail below:

1) Interconnection: The connections between operations reflect the local circuit complexity. As introduced in Section III-A2, the edge weight in our dependency graph denotes the number of wires that connect the operators. For each operation, we compute the number of edges and sum the edge weight to represent the interconnection information, such as the number of predecessors and fan-in. Among all the connections of one operator, their contributions to fan-in/fan-out may vary, so we extract the connection with the maximum number of wires and compute its edge weight percentage of the total fan-in/fan-out. To provide a more comprehensive representation of the local relationships among operations, we further expand the scope and extract similar features after including the two-hop neighbors of each node in the graph.

2) Resource: Different kinds of resources on FPGA, i.e., DSP, BRAM, LUT and FF, occupy different locations of the device and hence constrain the placement of operations. Therefore, the resource-related information of operations and their surrounding operations for each resource type need to be considered as shown in Table II. Resource usage denotes the number of resource units consumed by each operation. Resource utilization ratios are computed by comparing the resource usage of one operation with the available resources on FPGA and the resource usage of the function in which this operation belongs to. Similar to the interconnection category, we extract resource-related features for each operation and compute the total resource usage and utilization ratios of the node's predecessors/successors. The two-hop neighbors of each operator are also considered.

3) $\frac{\#Resource}{\Delta T_{cs}}$*:* Besides resource-related metrics, the impact of surrounding operators is also related to the spatial distance. Take an operation Op with two different successors S_1 and S_2 as an example, if S_1 has to process the result of Op immediately after the computation finished, while S_2 is executed several clock cycles later, the distance constraints

TABLE II. LIST OF FEATURES

Category	Feature Descriptions
Bitwidth	Bitwidth of each operation.
Inter-connection	Fan-in and fan-out of each operator and their summation; #predecessors, #successors and the summation; The max. number of wires among all the connections to one-hop neighbors and its percentage of the total fan-in and fan-out; Corresponding features after including two-hop neighbors.
Resource (for each resource type)	Resource usage and utilization ratios of each operation; The total resource usage and utilization ratios of all the predecessors and successors, and their summation; The max. resource usage and the corresponding percentage among all the one-hop neighbors; Corresponding features after including two-hop neighbors.
Timing	Delay(ns) and latency (clock cycles) of each operation.
$\frac{\#Resource}{\Delta T_{cs}}$	Resource usage and utilization ratios of predecessors/successors, divided by the subtraction of control states ΔT_{cs}; Corresponding features for two-hop neighbors.
Operator Type	The operation type of each operator, e.g., add, mul, xor, select; The number of each kind of operations among one-hop neighbors.
Global Information	Resource usage of the top-level function (F_{top}); Resource usage of the function in which the operation is located (F_{op}) and the corresponding percentage of the resources of F_{top}; Target/estimated clock period and clock uncertainty of F_{top} and F_{op}; Memories: #words, #banks, #bits and #primitives(words*bits*banks); Multiplexers: number, resource usage, input size and bitwidth.

between *Op* and its two successors are different given the same target clock period. S_2 is allowed to be connected to *Op* through longer wires. Even if S_1 and S_2 consume the same amount of resources, they have different influence on the local layout around *Op*. Therefore, we calculate $\frac{\#Resource}{\Delta T_{cs}}$ to evaluate the combined effects of resource usage/utilization ratios and timing information. Given a pair of dependent operations and suppose the preceding operator outputs the computation results at control state S_p and the subsequent operator begins to process the data at control state S_s, ΔT_{cs} is the subtraction of S_p and S_s based on the scheduling information as introduced in Section III-A2. Similarly, we also compute this feature for the two-hop neighbors of each operation.

4) Global information: Global features reflect the possibility of congestion, such as the available resources on FPGA and resource usage of functions. We extract the resource and timing related metrics of the top-level function and the function in which the operation is located. We also extract the information of memories and multiplexers used in the functions because both of them are important resource components that influence the layout. Note that the total number of available resources on FPGA are not listed as features in Table II, because we focus on the same FPGA device currently. We consider the impact of the available FPGA resources when computing the utilization ratios in the resource category in Table II.

C. Model Training

1) Sample Filtering: When a loop is unrolled, multiple copies of the same operation will be generated and mapped to different hardware units, which are placed to different locations on FPGA after PAR. If the unrolling factor is large, the location of replicas can be distant. For example, in *face detection* application [13], an unrolled loop generates 625 copies of the same set of operations, which are spread over the device. We find that parts of the replicas have similar features but their labels vary a lot because of their different physical locations. Figure 5 presents the distribution of vertical congestion metrics on FPGA for *face detection* and shows that lower congestion metrics are distributed at the margin of the device compared to

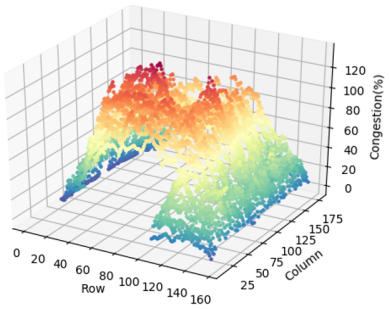


Fig. 5: Distribution of the vertical routing congestion metrics for *Face Detection* on FPGA.

the higher values in the middle of FPGA. When some replicas are placed around the margin of FPGA, the corresponding samples may contain very small labels, which deviate from most of the replicas and influence the estimation accuracy. These operations, called as marginal operations, taking up around 3.4 percent of all the operations in our benchmarks, should be filtered and removed. After filtering the outliers, the trained model can predict routing congestion more accurately. Another reason for filtering marginal operations in an unrolled loop is that our target is to detect the most congested region in the source code and the marginal operations with very small congestion metrics do not influence the detection.

2) Machine-Learning Models: The selection of machine learning models largely depends on the problem to be solved and the data that is used. Therefore, we train and compare various regression models to show their differences in prediction of routing congestion on FPGA as is evident in Table IV.

Linear Regression is used to model a target which is expected to be a linear combination of input features. In this work, we apply the *Lasso* linear model [15] with L1-regularization, which is to minimize the least-square penalty on the training data. The tuning parameter of the *Lasso* model is a constant parameter that multiplies the L1-regularization term and determines the sparsity of model weights.

Artificial Neural Network (ANN) is a promising technique to model non-linear relationships in the dataset [16]. Between the input and output layers, there are several hidden layers in which each neuron performs a weighted linear transformation on the values from the previous layer, followed by a non-linear activation function. It is more challenging to train the ANN model because a number of hyperparameters need to be tuned carefully and thus it takes a longer time to train.

Gradient Boosted Regression Trees (GBRT) combines multiple weak prediction models to form a powerful regression ensemble, which is an accurate and effective learning method [17]. GBRT builds the model in a stage-wise manner and introduces a weak estimator in each stage based on the gradients of the existing weak estimators. Several parameters require to be tuned such as the number of estimators and the learning rate that scales the step length of the gradient descent.

D. Resolving Congestion

Based on our trained models, routing congestion can be predicted and the most congested part of the source code can be recognized. Typically, it is easier to resolve congestion problems at the early design cycle. Through early detection, congestion can be resolved in advance without going through the downstream RTL implementation flow. There are several

TABLE III. PROPERTY SUMMARY OF BENCHMARKS

Metrics	WNS(ns)	Freq.(MHz)	Vertical Cong(%)	Horizontal Cong(%)	Avg. (V, H) (%)
Max	-3.253	75.5	133.33	178.96	144.87
Min	-13.643	42.3	5.06	8.90	6.73
Avg.	-8.386	54.4	60.58	72.47	64.89

methods to resolve routing congestion in HLS, such as modifying the code structure of the design and selecting suitable HLS directives. In Section IV-C, we will present a detailed example and demonstrate how to resolve routing congestion problems easily at the source code level.

IV. EXPERIMENTAL RESULTS

We implement our approach in Python and train the models with the scikit-learn [18] machine learning library. Our dataset is built from the Rosetta [13] benchmark suite, which contains six fully-developed realistic and optimized applications for HLS-based FPGA design. To fully utilize the available resources on FPGA and investigate the influence of routing congestion, we combine several benchmarks within the same top function and run the complete HLS and RTL implementation design flow to obtain the congestion metrics after PAR. Specifically, *Face Detection* is complex enough to occupy the device and thus is tested individually. *Digit Recognition* and *Spam Filtering* are invoked by the same function and the rest three applications, namely, *BNN*, *3D Rendering* and *Optical Flow*, are tested in an integrated function. We back trace the vertical and horizontal congestion metrics per CLB to the IR operations of each design, extract related features for each operation and build our dataset which consists of 8111 samples totally. Table III summarizes the RTL implementation results of the benchmarks, including performance and routing congestion metrics. The average of vertical and horizontal congestion metrics in the last column is the mean value of the two metrics for each CLB. Routing congestion metrics denote the estimated utilization percentage of routing resources in the vertical and horizontal directions of the tiles on FPGA and a utilization rate of over 100% indicates the potential routing problem around that region. All the designs are synthesized and implemented with Xilinx Vivado Design Suite 2018.1, and the target FPGA device is Zynq XC7Z020CLG484. The target frequency is set to 100 MHz. All experiments are conducted on an Intel Xeon CPU running at 2.5 GHz.

A. Estimation Accuracy

We randomly select 80% samples from our dataset for training and the rest 20% for testing. We employ a 10-fold cross-validation on the training set and grid search is applied to find the best hyperparameters of each model. The testing set is totally unseen and only used to evaluate estimation accuracy. To evaluate the congestion estimation accuracy, we compute the mean absolute error (MAE), defined as $\frac{1}{N}\sum|y_i - \hat{y}_i|$, and the median absolute error (MedAE), defined as $median(|y_1 - \hat{y}_1|, ..., |y_n - \hat{y}_n|)$. $\hat{y}_1$, $\hat{y}_i$ and $\hat{y}_n$ are the predicted values of the first, the i-th and the last samples and y_1, y_i and y_n are the corresponding actual values. N is the size of the testing set. MAE measures the average value of the absolute relative errors and MedAE reflects the distribution of the absolute relative errors which is robust to outliers [18].

TABLE IV. CONGESTION ESTIMATION RESULTS

Regression Models		Vertical Cong (%)		Horizontal Cong (%)		Avg. (V, H) Cong (%)	
		MAE	MedAE	MAE	MedAE	MAE	MedAE
Not Filtering	Linear	13.90	10.88	18.02	12.63	13.73	9.94
	ANN	12.19	7.91	17.68	12.62	12.27	8.17
	GBRT	10.55	7.37	15.71	10.89	10.57	6.78
Filtering	Linear	12.41	9.20	17.48	12.16	12.76	9.50
	ANN	10.23	7.43	16.61	11.78	11.67	7.83
	GBRT	9.59	6.71	14.54	10.05	9.70	6.81

Table IV compares the estimation accuracy of different models for routing congestion prediction. The GBRT model outperforms the ANN and the linear regression models and predicts all the three congestion metrics with the highest accuracy. As discussed in Section III-C1, we filter the marginal operations with the large deviation from other samples in an unrolled loop. As shown in Table IV, filtering further improves the estimation accuracy for each model and reduces the MAE of the GBRT model to 9.59%, 14.54% and 9.70% for the estimation of vertical, horizontal and their average routing congestion metrics, respectively. The smaller values of MedAE indicate that our models especially the GBRT model can predict routing congestion more accurately for at least half of the operations. Since we attempt to locate the most congested region in the source code, the accuracy of our model is sufficient to solve our problem and guide the subsequent optimization in HLS. In physical design, routing congestion is modeled to guide the placement with high accuracy [5, 6], because informative physical metrics can be obtained during the placement stage such as the number of bounding boxes, the net cuts per region and the pin count within a gcell. They require a highly accurate model to predict the routability, decide the location of each element and improve the quality of their placement algorithms. Compared to the works in physical design, the estimation error of our model is larger but acceptable due to the lack of detailed physical information and the wide gap of the abstraction level between HLS and the post-implementation after PAR. The cumulative optimization, imposed by each step in the downstream RTL implementation flow, is challenging to be captured at the source-code level.

B. Feature Importance

To clarify and interpret different effects of our features on the routing congestion estimation, we assess the importance of different categories of features through the GBRT model. The GBRT model measures the feature importance by averaging the number of times that a feature is used as a split point of the trees in the ensemble model. Table V lists the most important categories of features for each congestion metric. The order of the feature categories in each column is determined by the ranking of importance. We can see that $\frac{\#Resource}{\Delta T_{cs}}$ has the greatest influence on both vertical and horizontal congestion metrics and also affect the average congestion metric a lot. This is expected because $\frac{\#Resource}{\Delta T_{cs}}$ reflects the combined impact of resource consumption and timing constraints and indicates the degree of crowding around operators. We also find that among the features of $\frac{\#Resource}{\Delta T_{cs}}$ and resource categories, the features related to FF and LUT for the two-hop neighbors as explained in Section III-B exert greater influence. The interconnection category captures the complexity of connections between

TABLE V. IMPORTANT FEATURE CATEGORIES

Metrics	Vertical Congestion	Horizontal Congestion	Avg. (V, H) Congestion
Important Feature Categories	$\frac{\#Resource}{\Delta T_{cs}}$ Resource Interconnection Global (Mux)	$\frac{\#Resource}{\Delta T_{cs}}$ Resource Interconnection Global (Memory)	Resource $\frac{\#Resource}{\Delta T_{cs}}$ Interconnection Global (Mux)

TABLE VI. CASE STUDY: PERFORMANCE IMPROVEMENT

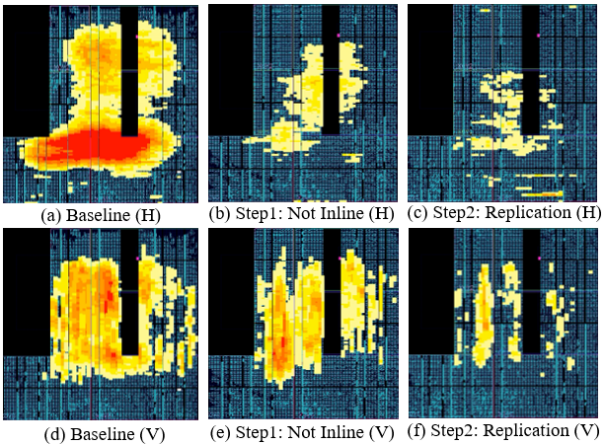
Implementation	WNS (ns)	Max Freq. (MHz)	Δ*Latency* (cycles)	Max Cong Vert, Hori (%)	#Congested CLBs ($\geqslant$ *100%*)
Baseline	-13.643	42.3	1.08e+6	133.33, 178.96	1272
Not Inline	-3.504	74.1	+23	129.85, 97.60	193
Replication	-0.767	92.9	+0	106.15, 104.73	17

dependent operators, which directly impact the local layout. Global information indicates the overall performance of a design. Specifically, the related information of multiplexers and memories has a greater effect on congestion estimation than other global features.

C. Case Study

When optimizing an HLS-based application, the trade-off relationship has to be considered carefully. As shown in Section II, *Face Detection* suffers from severe routing congestion problem which degrades the maximum frequency significantly even though the latency is reduced. In this subsection, we illustrate how to improve the maximum frequency of *Face Detection* and keep a low latency at the same time by resolving congestion. Based on our trained model, we can locate the most congested region and resolve congestion at the source-code level without running the RTL implementation flow.

The results of performance improvement are shown in Table VI. The baseline is the original design applied with several optimization techniques, and "Not Inline" and "Replication" denote our methods applied in the first and second steps to reduce congestion respectively. The horizontal (H) and vertical (V) congestion maps in each step are presented in Fig. 6 and the color represents the same meaning as in Fig. 1. In *Face Detection*, each image unit goes through the cascade classifier function which contains multiple stages of classifiers. The cascade classifier function and all the classifiers within it are inlined to reduce the latency. However, this incurs the issue since *function inlining* increases the complexity in C synthesis and generates a larger design. Routing congestion is detected at the region where multiple results returned by the classifiers are summed up and compared. Therefore, the first step is to remove *function inlining* which significantly reduces the maximum congestion metrics and the number of congested CLBs, as shown in Table VI. The performance is also improved with a higher maximum frequency and a slightly increased latency. After the first step, large routing congestion metrics are detected at the inputs of classifiers. This is because all the classifiers access data from the same completely partitioned array and multiple classifiers share the same inputs, leading to a large number of interconnections. To reduce the number of interconnections, the second step is to modify the high level source code by replicating the values of the input data and sending the copies to different classifiers. After that, the number of congested CLBs with over 100% congestion metrics is reduced to 17 and the maximum frequency is increased to 92.9MHz, while the latency remains unchanged compared to the step one. By modifying the source code or selecting



(a) Baseline (H) (b) Step1: Not Inline (H) (c) Step2: Replication (H)
(d) Baseline (V) (e) Step1: Not Inline (V) (f) Step2: Replication (V)

Fig. 6: Resolving Routing Congestion of *Face Detection*

suitable HLS directives, routing congestion is resolved easily and the performance is improved significantly, demonstrating that early detection of routing congestion in HLS is of great importance to enhance the developing efficiency.

V. CONCLUSION

In this paper, we propose a novel machine-learning based method to predict routing congestion for FPGA high-level synthesis. After back tracing congestion metrics to IR operations, we extract informative features and train three different machine learning models for congestion estimation. Experiments show that our GBRT model achieves the highest prediction accuracy. Based on the model, routing congestion can be reduced easily and performance is improved significantly.

REFERENCES

[1] J. Zhao *et al.*, "Comba: a comprehensive model-based analysis framework for high level synthesis of real applications," in *ICCAD*, 2017.
[2] S. Dai *et al.*, "Fast and accurate estimation of quality of results in high-level synthesis with machine learning," in *FCCM*, 2018.
[3] H. Zheng *et al.*, "Fast and effective placement and routing directed high-level synthesis for fpgas," in *Proc. of FPGA*, 2014, pp. 1–10.
[4] A. Belghadr *et al.*, "Metro-on-fpga: A feasible solution to improve the congestion and routing resource management in future fpgas," *INTEGRATION, the VLSI journal*, vol. 47, no. 1, pp. 96–104, 2014.
[5] G. Chen *et al.*, "Ripplefpga: Routability-driven simultaneous packing and placement for modern fpgas," *TCAD*, 2017.
[6] D. Maarouf *et al.*, "Machine-learning based congestion estimation for modern fpgas," in *Proc. of FPL*, 2018.
[7] W. Li *et al.*, "Utplacef: A routability-driven fpga placer with physical and congestion aware packing," *TCAD*, vol. 37, pp. 869–882, 2018.
[8] J. Wu *et al.*, "Congestion aware high level synthesis combined with floorplanning," in *PACIIA*, vol. 2, 2008, pp. 935–938.
[9] W. E. Dougherty *et al.*, "Unifying behavioral synthesis and physical design," in *Proc. of DAC*, 2000, pp. 756–761.
[10] Y. Wang *et al.*, "Reallocation and rescheduling after floor-planning for timing optimization," in *Proc. of ASIC*, vol. 1, 2003, pp. 212–215.
[11] J. Cong *et al.*, "Towards layout-friendly high-level synthesis," in *Proc. of ISPD*, 2012, pp. 165–172.
[12] M. Tatsuoka *et al.*, "Physically aware high level synthesis design flow," in *Proc. of DAC*, 2015, p. 162.
[13] Y. Zhou *et al.*, "Rosetta: A realistic high-level synthesis benchmark suite for software programmable fpgas," in *FPGA*, 2018, pp. 269–278.
[14] P. Coussy *et al.*, "An introduction to high-level synthesis," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 8–17, 2009.
[15] G. A. Seber *et al.*, *Linear regression analysis*, 2012, vol. 329.
[16] B. Yegnanarayana, *Artificial neural networks*, 2009.
[17] J. Elith *et al.*, "A working guide to boosted regression trees," *Journal of Animal Ecology*, vol. 77, no. 4, pp. 802–813, 2008.
[18] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.