Early Identification of Timing Critical RTL Components using ML based Path Delay Prediction

Prianka Sengupta
Texas A&M University
College Station, USA
prianka.sengupta@tamu.edu

Aakash Tyagi
Texas A&M University
College Station, USA
tyagi@cse.tamu.edu

Yiran Chen
Duke University
North Carolina, USA
yiran.chen@duke.edu

Jiang Hu
Texas A&M University
College Station, USA
jianghu@tamu.edu

2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD) | 979-8-3503-0955-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/MLCAD58807.2023.10299879

***Abstract*—In chip design, it is crucial to identify timing critical components early on to preemptively fix any timing issues and avoid numerous design convergence iterations. However, obtaining this information requires one to run the time intensive physical design flows (synthesis, placement, etc.). To this end, we propose a machine learning approach to predict timing path delays at a granular level directly from RTL design, thereby avoiding the reliance on synthesis and placement. This will allow designers to quickly evaluate the delays of timing critical paths as well as the worst-case delay at an early stage. Experiment results show that our approach predicts timing path delays with 91% accuracy when compared with post-placement timing analysis. Furthermore, this approach identifies the specific logic sections in an RTL code responsible for the longest timing delay paths and is more than 40 times faster than the conventional synthesis and placement.**

***Index Terms*—RTL Design, Static Timing Analysis, Machine Learning**

I. Introduction

A typical digital design flow can be simplified to the following steps - development of register-transfer-level (RTL) code, logic synthesis of the RTL, followed by placement and routing (PnR). Static timing analysis of the gate level netlist is performed multiple times at various post-synthesis stages to identify timing failures and to explore possible solutions for timing convergence. One of the solutions to address timing issues is to change EDA tool recipes to optimize timing paths in the design. However, this approach is iterative and can lead to sub-optimal trade-off between timing, area and power [1]. Another effective approach is to identify the RTL source code responsible for failed timing paths and attempt to modify these RTL sections. Often times, a combination of both recipe changes and RTL improvements is required to resolve timing failures. However, RTL source code optimization is resource expensive and is also iterative in nature since evaluating the timing improvements depend on timing results from post-synthesis stages. Moreover, the time required for these RTL changes leaves little time for design verification in constrained project schedules. Due to these challenges, RTL modifications are often avoided, even though it is one of the more effective options for minimizing late stage timing fixes.

In many cases, improving the RTL code exhibits significant performance improvements and assists in solving timing issues. For example, adding pipeline stages to split the logic between two registers helps break down long delay paths into multiple shorter paths. This reduces the logic levels and hence resolves critical timing path issues. Large path delays can also be addressed by register re-timing, which moves registers in the logic network to optimal locations to reduce path depth and delays. For high fan-out scenarios, duplication of logic in RTL source code can improve routing delay and arrival time. Therefore, identifying and resolving the timing critical paths at RTL development stage is imperative for not only the RTL designers but also for accelerating entire design cycle.

In this work, we introduce a machine learning (ML) approach, particularly a tree-based ensemble method (XGBoost [2]) to predict the timing critical paths at the RTL stage without having to rely on timing feedback from synthesis and PnR results, allowing early edits of RTL to address critical timing issues. This work extracts features directly from RTL code (Verilog) to use as inputs to a ML model to make timing predictions at a granular design level. The ML model aims to provide predictions of individual timing path delays which enables the designers to focus on the problematic areas where timing failures may appear during the implementation stages. The contributions of this work include the following:

- Classify the timing paths of any RTL design in bins of ascending delay classes.
- Predict the worst case timing path delay for a design.
- Make predictions to identify logic components in the RTL source code that will generate timing critical paths.
- Provide these results more than 40 times faster than the commercial EDA tools.

The results on a set of benchmark designs show that our approach can reach an average of 91% accuracy in predicting post-placement timing path delays. To the best of our knowledge, this is the first work which directly extracts features from RTL designs to predict timing as well as identify RTL code sections that's responsible for the timing failures.

The remainder of the paper is organized as follows. Section II discusses existing academic literature related to delay prediction of timing paths. Section III presents the primary goals of this work, followed by Section IV describing the general concept behind our approach. The details of our novel RTL parsing and feature extraction technique is described in Section V. The Machine Learning aspect of our work in discussed in Section VI, followed by experimental results and observations in Section VII. We present few real world applications of our proposed technique in Section VIII that shows the benefits of early identification of timing issues in an RTL design.

II. Related Work

There are few works which utilize machine learning to predict timing at different stages of the design process. In recent works [3] and [4], timing prediction models are developed based on RTL designs consisting of primitive logic gates. The trained ML models are tested using RTL of various adder architectures to evaluate the prediction accuracy of the pin to pin delay. However, these works only address structural RTLs consisting of primitive combinational logic gates and doesn't explore real world hierarchical and behavioral RTL designs. In addition, the prediction model uses features from Static Timing Analysis (STA) reports generated using synthesized netlists and doesn't consider effects of cell placement.

Another recent work [5] aims to predict the power, performance and area outcomes using RTL features generated from Yosys [6]. The path delay prediction model in this work is trained with a small dataset that contains a mixture of few real path samples as well as synthetically generated samples. To accelerate the processing time, this work samples and aggregates the timing paths to report the delay of the worst case timing path for a given design, which does not provide visibility of the layers of timing issues that may arise from sub-hierarchies of a design. Also, the machine learning prediction results in this work reports synthesis timing results and does not consider effects of placement and layout. The commercial tool in [7] provides a high level overview on critical timing paths from the RTL modules which helps designers gauge the possible implementation effects in advance. This tool uses calibrated models to make timing estimations, although the runtime still ranges from hours to days for finding the optimal solutions. Another relevant work [8] uses machine learning to predict post-placement dynamic power and total negative slack results for a given RTL design, where it utilizes Verilog RTL code and synthesis recipe parameters to compute features for the prediction task. However, this work predicts total negative slack for the entire RTL design and doesn't predict individual timing path delays of the RTL design.

In general, all the previous works on ML based timing prediction in the ASIC design domain have different goals and solution approaches. There is no previous work that aims to provide quick and early identification of the timing critical components in the RTL code which utilizes features collected directly from the RTL source code to train ML models, to the best of our knowledge.

III. Problem Formulation

A. Prediction of timing path delays from RTL

The primary goal of the proposed approach is to predict post-placement path delays using a machine learning (ML) model for all possible timing paths from an RTL. The timing paths identified in the RTL for delay prediction are between input ports to registers, registers to registers and register to output ports. The technique of identifying these timing paths in the RTL and extracting relevant features are key contributions of this work. The ML model is trained with large sets of path data, including the feature vector for each path and it's target value. The proposed method predicts the delay of each timing path, including paths that are likely to encounter setup time failures.

B. Locating sources of the worst delay paths in RTL

Leveraging the timing path delay prediction achieved in the first goal, the path with the worst case delay can be easily traced in the source RTL. In addition, our approach highlights specific lines of Verilog codes with the registers as well as corresponding sections of the logic that are responsible for the timing paths with the worst case delay.

IV. Overview

The entry point of our approach is the Verilog RTL source code of a given design. The RTL is first converted to its Abstract Syntax Tree (AST) as explained in Section V, followed by a novel AST post-processing method to construct a directed graph G, as well as to compute features. Figure 1 shows a high level overview of our approach. Graph G consists of nodes that represent the behavioral logic, registers and input/output (IO) ports, connected by directed edges to define the data-flow.

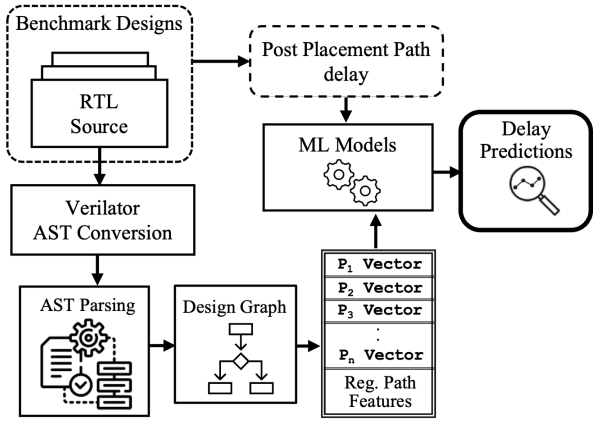

Fig. 1. Overview of RTL processing and timing path delay prediction

The nodes in graph G can be categorized to two types: the first being variable nodes (V_r) that represent IO ports and registers, and the second type for nodes that represent behavioral logic (V_b). The set of all directed edges in G is denoted by E. Once G is constructed, all paths between node pairs (u, v) are identified, where $(u, v) \subset V_r$. Each of theses path consists of a set of edges e_p, where $e_p \subset E$ and a set of nodes v_p where $v_p \subset V_b$. A feature vector is computed for each path, taking into account the various attributes of e_p and v_p. The details of this feature computation can be found in Section V-B. A set of benchmarks designs from various domains with varying sizes (listed in Section VII-A) were used for obtain these behavioral register paths and their associated feature vector. A collection of these timing paths and their feature vectors along with the reference post-placement delays are provided as inputs to the ML model for training. In order to collect the the reference delays the benchmark designs are taken through synthesis and placement steps using commercial implementation tools that report the delay of individual timing paths that we use as target labels in the training process. The path delays are categorized into k ascending delay bins in order to represent the learning task as a classification problem, where k is a specifiable parameter.

V. RTL Feature Extraction

Verilog is one of the most prominent hardware description languages (HDL) used for hardware design. It uses constructs that are similar to non-HDL programming languages to build logic and storage elements. We took motivation from the compilation process of general programming languages, where the source code is often converted to an Abstract Syntax Tree (AST) representation. The AST of a given source code is a tree-like data structure that represents the syntactical components of the code in a formal hierarchical format for systematic post processing and feature extraction.

A. Verilog Abstract Syntax Tree (AST)

Multiple open-source tools exist that can generate an AST from Verilog designs. Some notable ones are Verilator [9], PyVerilog [10], Verible [11] and Yosys [6]. The AST of a design contain a tree structure that follows the hierarchy established in the source code. The tree starts from a root element (e.g. top module) and ends in many leaf nodes (registers or I/O ports). In this work, we used Verilator to process Verilog designs and generate the AST in XML format, which is a common markup format used to represent hierarchical data. An example of XML-based AST description is shown in Figure 2. The example AST contains multiple hierarchical elements, each marked by a starting and ending tag such as `<module>`, `<always>`, `<add>` etc. Generally these element tag names represent their Verilog counterpart with the same name. For example, the `<module>` element in XML represent the top level Verilog module, the `<always>` element represent the always procedural block and so on. Some of the less obvious elements tags in the XML are:

- `<var>,<varref>`: IO port, parameter or register declaration or references.
- `<senitem>`: Sensitivity list for procedural blocks.
- `<cond>`: Conditional logic(*if*,*else-if*).
- `<gt>,<lt>,<eq>`: Greater than, less than or equal logical operators.
- `<assign>,<assigndly>`: Denotes value assignment to a variable (reg, wire or port).
- `<add>,<sub>,<mul>`: Addition, subtraction or multiplication operators.

The post-processing engine developed in this work processes the XML based AST construct a directed graph representing the design. An example output of the parser is the design graph shown in Figure 2. The nodes with rectangular outline represents the IO ports or registers and nodes with oval/circular outline represent the behavioral logic operations involving the IO or the register nodes. The edges represent the flow of data. For example, the *sub* node has two incoming edges from *data* and from a constant node. These edge connections originate from the subtraction operation seen in the Verilog source code where the constant value of 4'd2 is subtracted from the *data* variable. However, the subtraction operation only occurs when the parent *if* condition is true. Therefore the outgoing edge of the *sub* node travels through a *cond* node and ends at the *result* node. Design graphs created from the AST primarily shows

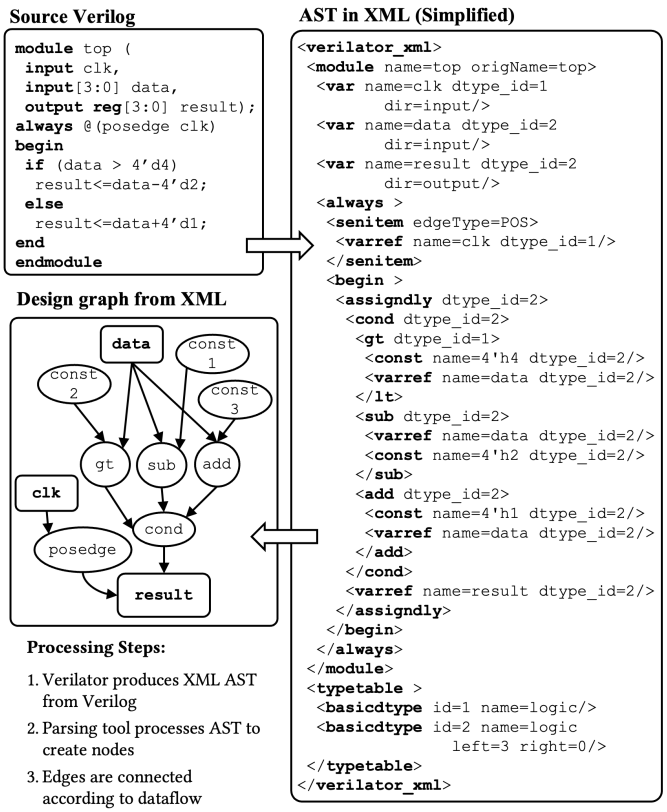

Fig. 2. Example of creating a design graph from Verilog via AST Conversion

the flow of data between IO ports or registers, controlled by conditional logic and arithmetic or logical operators.

B. Computing Path Features from Design Graph

In a design graph the path from one node to another consists of one or more edges and may travel through behavioral logic nodes. We refer to such paths as *behavioral paths* and denote the set of all behavioral paths in a design as P_b. Each path in P_b may carry multiple bits of data and pass through different logic blocks such as conditional elements, comparators, primitive logic and arithmetic logic. Examples of such behavioral paths can be seen in Figure 2, where behavioral paths from input nodes (*data*) to output nodes (*result*) travels through comparators, arithmetic and conditional nodes. The paths may also undergo concatenation, reduction operators or slicing, which can affect the total number of single-bit paths present in the design. The behavioral logic nodes along each path dictates the logic depth, cells types and delay of the actual gate level paths that will be constructed in the post-synthesis or post-placement stage. Hence, we define and compute the following path features that summarize various useful behavioral path properties: 1) Behavioral logic depth along the path; 2) Average logic input/output (I/O) bits along the path; 3) Count of conditional logic nodes (If/Case, Compare); 4) Count of arithmetic logic nodes (add/sub,mul,moddiv); 5) Count of basic logic nodes (and, xor, negate, etc.); 6) Number of logic nodes shared with other paths; 7) Number of logic nodes unique to the path; 8) Average cell complexity score n^2, where n is the number of logic I/O bits. The extracted features, along

with the starting and ending sequential node names are saved as a feature vector for training the machine learning model.

VI. Machine Learning-Based Prediction

The features extracted from RTL are fed into the machine learning models in both training and inference. In parallel to the feature data, the training process also requires ground truth data as labels, which are obtained from post-placement timing analysis as mentioned in Section VII-A.

A. Predicting the Delay of Timing Paths

Prior to defining the learning task of the delay classification, profiling the delay value distribution of the ground truth data is required to assess the practical range of delay values that need to be predicted. After filtering out the outliers, distribution of the path delays is observed which is categorized in a reasonable number of k classes as shown in Figure 3. Therefore, each path in the design is labeled as one of these k delay classes. Hence, this becomes a multi-class classification task that predicts the delay class for any given timing path in an RTL.

B. Model Configuration

For the machine learning classification, we employ the XGBoost infrastructure [2] to build and train a XGBoost classification model. The XGBoost model consists of 100 boosted trees with a maximum depth tree of 7. The dataset in this learning problem is structured in nature and tree-based ensemble methods are known to outperform more complex deep learning techniques for such data [12].

VII. Experiment and Results

A. Generating Training Data

For training the timing prediction models, it is required to collect a large set of data with labeled path delays. We employ 26 Verilog designs available in IWLS 2005 benchmark [13] and OpenCores, which are listed in table I. Each of these designs is synthesized and placed using industry standard commercial tools. During the synthesis and placement, the maximum achievable clock frequency for each design is used to ensure that the best efforts of the EDA tools are captured in the training data. At the post-placement stage, a detailed report of each path delay is collected from timing analysis reports of each design. Path delays from each design are labeled with the delay class categorization method. In our experiment, labeling the paths in $k = 8$ classes based on the magnitude of their delay provides good balance between model complexity, training and inference time. A total of 161K path data is collected from the benchmark designs and Figure 3 shows the corresponding logarithmic histogram of the path delays.

The delay classes from A to H represent path delays in an increasing order, with each class having a spread of 3.5nS. The distribution shows that classes with higher delays contain relatively lower number of paths. This is expected since paths with lower delays are more common in typical digital designs. However, this causes our classification data set to be imbalanced which had be be compensated in training step using the stratified sampling technique [14]. Stratification ensures

TABLE I
26 Benchmark designs used for training and testing

Training Designs	Seq. Cells	Comb. Cell	Total Gates
ss_pcm	87	173	260
uart2bus	157	611	768
wb_dma	523	1,728	2,251
mem_ctrl	1,065	3,296	4,361
⋮	⋮	⋮	⋮
fpu	663	31,881	32,544
xge_mac	13,301	33,366	46,667
scdma_viterbi	68,393	164,236	232,629

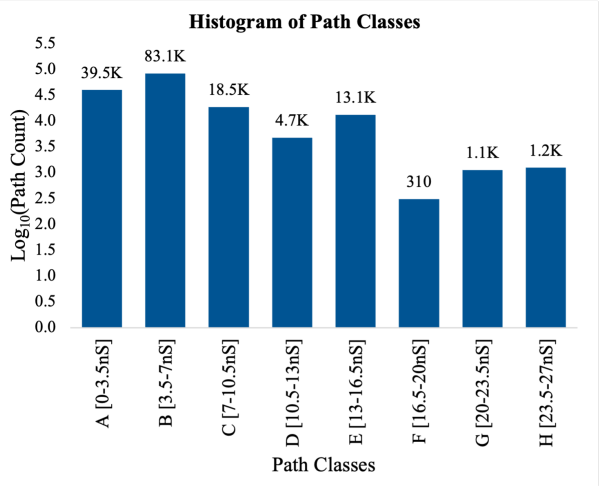

Fig. 3. Histogram of path delays and classes in the training data

that the samples for each class has a similar distribution, which helps prevent biased model training and evaluation. The maximum range of the distribution (class H) ensures that worst case delays of all training designs are included for training. The collected path delays were split into training the testing data set using a 75:25 ratio. Testing data was kept separate from the training sequence in order to collect the prediction accuracy for timing path unseen to the trained model. Once trained with data from a given technology node, the model can be applied for any unseen design targeted for the same process technology. For application in a new technology node, the ML models will require retraining with training data generated using the respected process technology.

B. Path Delay Prediction

The XGBoost classification model was trained with 75% of the 161k timing path data. For the 8-class timing prediction task, an average classification accuracy of 91% was achieved, along with a precision of 88.3%, recall of 86.2% and and F1 score of 87.2%, where accuracy, precision, recall and F1-score are defined by:

$$Accuracy = \frac{TruePositive + TrueNegative}{(TotalPredictions)} \quad (1)$$

$$Precision = \frac{TruePositive}{(TruePositive + FalsePositive)} \quad (2)$$

$$Recall = \frac{TruePositive}{(TruePositive + FalseNegative)} \quad (3)$$

$$F1\text{-}Score = 2 * (\frac{Precision * Recall}{(Precision + Recall)}) \quad (4)$$

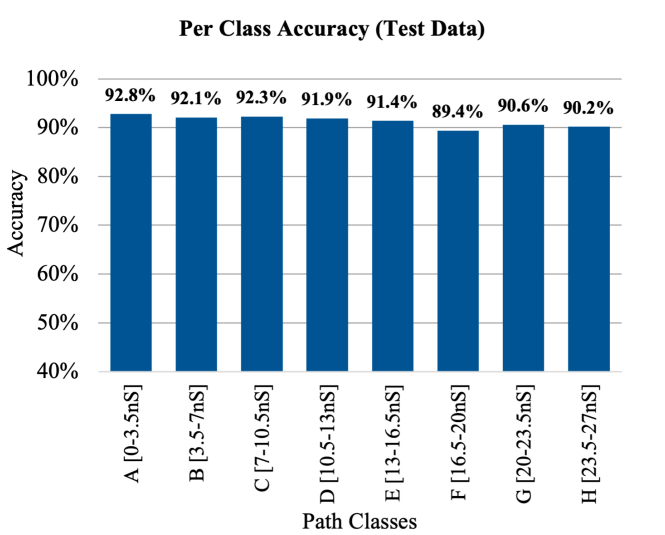

Fig. 4. Per-class accuracy of delay prediction

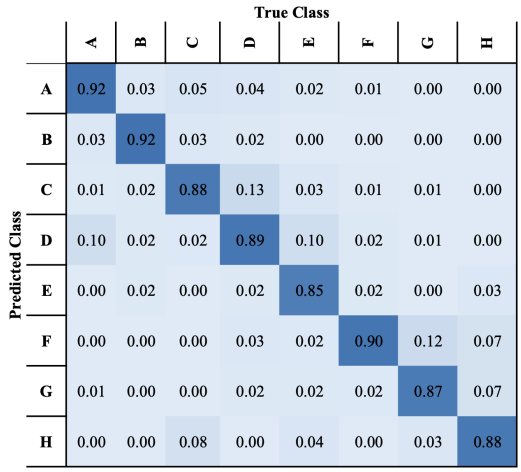

Fig. 5. Confusion matrix for predicted class vs true class

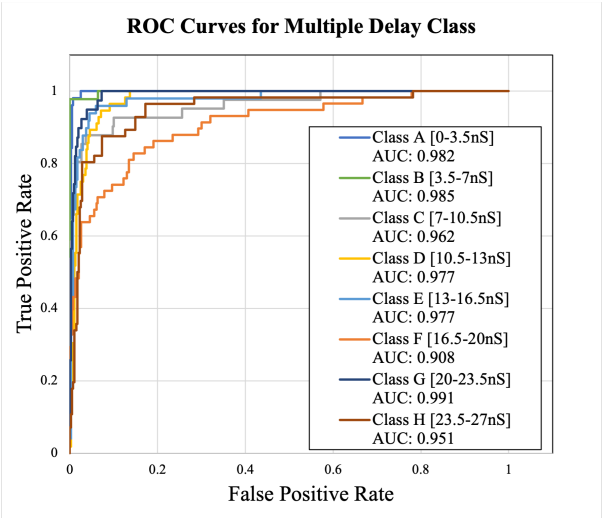

Fig. 6. Receiver Operator Characteristics for the XGBoost multi-class path delay classification model

The accuracy per class is shown in Figure 4 followed by the confusion matrix of true delay class vs predicted delay class in Figure 5. The confusion matrix represents the rate of correct predictions, where the predicted class matches the actual class. The path delay classifier exhibits high rate of correct predictions, represented by the values along the diagonal fields of the matrix. The receiver operator characteristics curve for the trained multi-class classifier is shown in Figure 6 along with the Area Under the Curve (AUC) for each class. A ROC curve illustrates the trade-off between the true positive rate (TPR) and the false positive rate (FPR) for various classification thresholds [15]. The AUC for each class is a metric that summarizes the classification performance of a classifier with 1 being the best AUC possible. The path delay classifier achieves AUC greater than 0.9 for all classes with the lowest score for Class F, which is attributed by the low number of training samples available in that class.

C. Training and Prediction Runtime

To quantify the quick time-to-result from the proposed approach, we compare the time it takes to receive timing feedback from post-placement stage of implementation against the time required by our method to infer path delay from RTL in Table II. Multiple benchmark designs with varying path counts are listed in the table. Although collecting the training data and training process initially consumes multiple hours, it is easily amortized by the speedup achieved from the proposed ML model. Furthermore, the generic nature of our ML model ensures that once trained, it can be used to predict delays for timing paths extracted from any unseen RTL design without the need for retraining. The synthesis, placement and ML inference times in Table II were collected on a 10 Core Intel(R) Xeon(R) CPU E5-2680 (2.80GHz).

TABLE II
RUNTIME COMPARISON OF THIS WORK WITH TRADITIONAL APPROACHES

Design Name	Number of Paths	Time required for path delay	
		Post-Placement[1]	*This Work*[2]
fpu_double	5273	118 minutes	3 minutes
des3_perf	9096	153 minutes	3 minutes
scdma_viterbi	68091	462 minutes	9 minutes

[1]Time required for synthesis, placement and reporting of path delays
[2]Time required for RTL path feature extraction and ML inference

VIII. Applications

A. Identifying and fixing timing hot-spots in RTL

In this section we explore some practical application of the proposed method in the context of developing a Verilog RTL design. The register delay paths for a Verilog design with various combinational expressions were analyzed to identify its worst case path, followed by an attempt to optimize the relevant RTL code section based on the prediction from our approach. Synthesizing the unmodified RTL design with a strict clock period constraint in a commercial implementation flow yielded a worst path, the start and end points of which are listed in the timing reports. The report from the commercial tool also lists standard cells that are a part of the worst path along with the total path depth. The time to generate detailed timing reports through logic synthesis and layout requires several hours to days, depending on the size of the RTL design. Also such reports usually lack the details that designer

Original RTL

```
126: case(state)
127: 3'b010: begin
128:  c_enable <= stall & (|(p_active ^ request));
129:  count <= value + 32'd1;
130:  next_state <= 3'b011;
131: end
```

Flow	Critical Delay Path
Post-Placement	**startpoint**: request, **endpoint**: c_enable **delay**: 3.96 nS (Class B)
This Work	**startpoint**: p_active, **endpoint**: c_enable **delay**: Class B, **Code Line**: 128

Modified RTL

```
126: case(state)
127: 3'b010: begin
128:  c_enable <= stall & (p_slots);
129:  count <= value + 32'd1;
130:  next_state <= 3'b011;
131: end
```

`p_slots` is optimized version of `|(p_active^request)`

Flow	Updated Delay Path
Post-Placement	**startpoint**: request, **endpoint**: c_enable **delay**: 3.36 nS (Class A)
This Work	**startpoint**: p_active, **endpoint**: c_enable **delay**: Class A, **Code Line**: 128

Fig. 7. RTL optimization based on timing feedback

needs to identify specific sections in the RTL code that require optimization for reducing the timing delay of the worst path. In contrast, our proposed approach is able to predict timing path delays 40 times faster than commercial tools for the design used in this experiment. The path reported from our approach not only provides the start and endpoints of the worst case path, it also indicates the behavioral nodes that appear along the worst case path. In the design graph created by our AST parser, each node conveniently stores the specific Verilog line number as an attribute. This helps the designer to trace back to the RTL code sections that needs optimization for improving timing. In the example shown in Figure 7, our method indicates the complex expression on line 128 to be the source of the critical path delay of 3.96nS. To replicate a fix, we simplify the expression to reduce the logic depth. The modified RTL exhibits a lower delay for the same endpoints, proving that the feedback provided by our approach is effective in quickly pinpointing the timing hot-spots in the RTL section.

B. Worst Case Delay Prediction

A timing critical path is often the cause of timing failure and results in the worst case delay during static timing analysis. Thus, identifying the specific path responsible for the worst case delay in a design is the first step to solving a multi-layered timing problem. Fixing the timing paths with negative slack is a multi-layer problem because a design that does not meet timing requirements typically contains a collection of failed paths ranked by their delay. Hence, solving the worst path is an iterative process as each step brings the next worst path into focus. Analyzing any design using our method allows us to quickly list the top n paths with the largest delay. Comparing these predicted worst paths against reference timing reports from the commercial placement tool shows that 90.4% of the predicted paths are correctly identified.

IX. Conclusion and Future Work

In this work, we have presented a machine learning based timing prediction method which utilizes a novel RTL feature extraction technique to identify granular level timing paths in RTL designs. The proposed approach is capable of classifying the delays of these timing paths and identifying the components in the RTL source code which contribute to the timing failures and the worst case delay. The results are achieved 40 times faster than the commercial approaches with an average accuracy of 91%. In future, we aim to explore regression based delay prediction models and incorporate the effects of various synthesis recipes on timing.

Acknowledgment

This work is partially supported by Semiconductor Research Corporation GRC-CADT 3103.001/3104.001 and National Science Foundation CCF-2106725/2106828.

References

[1] R. Puri, L. Stok, J. Cohn, D. Kung, D. Pan, D. Sylvester, A. Srivastava, and S. Kulkarni, "Pushing asic performance in a power envelope," in *Proceedings of the 40th annual Design Automation Conference*, 2003, pp. 788–793.

[2] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, 2016, p. 785–794.

[3] D. S. Lopera and W. Ecker, "Applying gnns to timing estimation at rtl," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022, pp. 1–8.

[4] D. S. Lopera, L. Servadei, V. P. Kasi, S. Prebeck, and W. Ecker, "Rtl delay prediction using neural networks," in *2021 IEEE Nordic Circuits and Systems Conference (NorCAS)*. IEEE, 2021, pp. 1–7.

[5] C. Xu, C. Kjellqvist, and L. W. Wills, "Sns's not a synthesizer: a deep-learning-based synthesis predictor," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 847–859.

[6] C. Wolf, "Yosys open synthesis suite," https://yosyshq.net/yosys/.

[7] J. Schultz, "Rtl architect: Parallel rtl exploration with unparalleled accuracy," Tech. Rep., June 2021.

[8] P. Sengupta, A. Tyagi, Y. Chen, and J. Hu, "How good is your verilog rtl code? a quick answer from machine learning," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022, pp. 1–9.

[9] W. Snyder, "Verilator and systemperl," in *North American SystemC Users' Group, Design Automation Conference*, 2004.

[10] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog hdl," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2015, pp. 451–460.

[11] T. Ansell and M. Saligane, "The missing pieces of open design enablement: A recent history of google efforts: lnvited paper," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2020, pp. 1–8.

[12] L. Grinsztajn, E. Oyallon, and G. Varoquaux, "Why do tree-based models still outperform deep learning on typical tabular data?" in *Advances in Neural Information Processing Systems*, vol. 35. Curran Associates, Inc., 2022, pp. 507–520.

[13] A. Mishchenko, S. Chatterjee, and R. Brayton, "Integrating logic synthesis, technology mapping, and retiming," in *Proc. IWLS'05*. Citeseer, 2006.

[14] A. Anand, G. Pugalenthi, G. B. Fogel, and P. Suganthan, "An approach for classification of highly imbalanced data using weighting and undersampling," *Amino acids*, vol. 39, no. 5, pp. 1385–1391, 2010.

[15] J. N. Mandrekar, "Receiver operating characteristic curve in diagnostic test assessment," *Journal of Thoracic Oncology*, vol. 5, no. 9, pp. 1315–1316, 2010.