# STREAMLINING TNS DATA COLLECTION FOR ML-BASED RTL QOR PREDICTION

An Undergraduate Research Scholars Thesis

by

PRANAV JAIN[1] AND KUNAL GUPTA[2]

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Aakash Tyagi

May 2023

Majors: Computer Science[1,2]

# RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

We, Pranav Jain[1] and Kunal Gupta[2], certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with our Faculty Research Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

# TABLE OF CONTENTS

Page

# ABSTRACT

Streamlining TNS Data Collection for ML-Based RTL QoR Prediction

Pranav Jain[1] and Kunal Gupta[2]
Department of Computer Science and Engineering[1,2]
Texas A&M University

Faculty Research Advisor: Dr. Aakash Tyagi
Department of Computer Science and Engineering
Texas A&M University

Chip designs must meet several requirements before they are ready for fabrication. One of these requirements is achieving convergence on timing (frequency). Meeting this requirement is a time-consuming task for chip designers in the industry for two reasons. First, the standard approach to procuring this metric involves running logic synthesis and placement, both of which can take hours to weeks on larger RTL designs. Second, since the timing requirement is rarely met after one design iteration, these processes need to be rerun multiple times to recalculate the metric to ultimately converge on the design's requirements. A critical measure of timing convergence is the total negative slack, commonly referred to by its acronym TNS. It indicates the sum of timing margins of all 'negative slack' paths that fail to meet the target clock cycle time. To expedite design convergence, our research team previously presented a machine learning-based approach to estimate the TNS values for chip designs expressed in Verilog hardware description language. This technique was orders of magnitude faster than running logic synthesis and placement on those same chips. In this work, we build on the previous approach by improving the initial data generation process. Getting "true" TNS values for training the machine

learning models involves running logic synthesis and placement with hundreds of synthesis recipes for each design, resulting in tens of thousands of synthesis and placement runs. Driven by the need to create a rich training data set, since new designs will be continuously added to the RTL developer's set of training designs, it behooves to reduce the number of synthesis and placement runs necessary to generate machine learning (ML) training data. By taking advantage of similarities in the distributions of TNS values across chip designs, the number of required synthesis and placement runs for *n* Verilog RTL designs and *m* unique synthesis recipes can be reduced from $O(nm)$ to $O(n+m)$ without meaningfully compromising the integrity of the training data and the accuracy of ML predictions. We present two methods for achieving this, both of which involve finding the common TNS distribution, then normalizing and computing missing values in the data set. The discoveries made by our research team have the potential to drastically reduce the time to market for a variety of semiconductor computing products, including but not limited to graphics processors, motherboards, and flash memory.

# ACKNOWLEDGEMENTS


## Contributors

We would like to thank our faculty advisor, Dr. Aakash Tyagi, our doctoral student advisor, Prianka Sengupta, and Dr. Jiang Hu for their guidance and support throughout the course of this research.

Thanks also go to our friends and colleagues and the department faculty and staff for making our time at Texas A&M University a great experience.

The AST parsing algorithm used in STREAMLINING TNS DATA COLLECTION FOR ML-BASED RTL QOR PREDICTION was designed and initially implemented by Prianka Sengupta.

All other work conducted for the thesis was completed by the students independently.

## Funding Sources

This undergraduate research received no funding.

# NOMENCLATURE

| | |
|---|---|
| ML | Machine Learning |
| RTL | Register-Transfer Level |
| HDL | Hardware Description Language |
| Verilog | Most widely used HDL for modeling RTL electronic systems |
| TNS | Total Negative Slack |
| AST | Abstract Syntax Tree |
| XGBoost | Extreme Gradient Boost |
| MSE | Mean Squared Error |
| PnR | Placement & Routing |

# 1. INTRODUCTION

## 1.1 Background

Chip designs need to meet several requirements before they are ready for fabrication. One of these requirements is convergence on timing (frequency), which can be measured by total negative slack, commonly referred to by its acronym TNS. TNS indicates the sum of timing margins of all ‘negative slack’ paths that fail to meet the target clock cycle time. For obvious reasons, a TNS of 0 is desired prior to sending the design for fabrication. Unfortunately, meeting the requirement of timing convergence is an effort intensive task for chip designers. Figure 1 illustrates the process that chip designers follow to meet design convergence targets. After the register transfer level (RTL) code has been written by designers, logic synthesis and placement are run on the source code to determine its TNS. If the result does not meet the TNS constraints of the final product, the designers must update the Verilog code, which is then passed to the synthesis and placement tools once again to determine the new TNS. This process repeats until the RTL code’s TNS converges to the required value. Since running logic synthesis and placement often takes hours or even days on larger RTL designs, significant lengths of time are wasted between iterations, making the design convergence task a lengthy process.

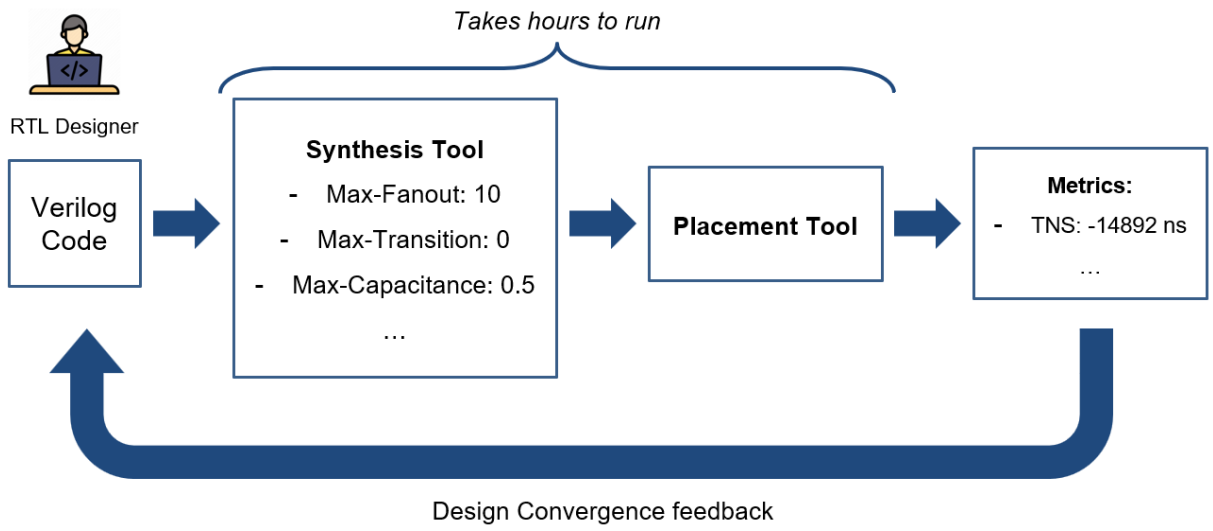


*Figure 1: Design Convergence*

To speed up the collection of TNS data in RTL chip designs, Sengupta introduced a machine learning-based approach that is orders of magnitude faster than running logic synthesis and placement [1]. The approach is illustrated in Figure 2. The technique starts by taking a Verilog RTL design and generating an Abstract Syntax Tree (AST) of the design using Verilator, a tool commonly used to convert RTL written in Verilog to a cycle-accurate behavioral model in C++ or SystemC. The generated AST is then passed through a novel parsing algorithm which examines several features of the AST and outputs the feature values. These values, along with several synthesis parameters, are formatted into a vector and used as inputs for machine learning models to predict TNS. The models used include linear regression, random forest, neural network, and XGBoost. The trained models achieve an average R-squared correlation value of 95%, with XGBoost performing the best for TNS prediction. All these models predict TNS six orders of magnitude faster than the one otherwise achieved by running synthesis and placement to compute the values directly. Getting accurate feedback this quickly on RTL code enables chip

designers to iterate on their designs faster and reach design convergence sooner, which greatly reduces the time to market for new computing products.

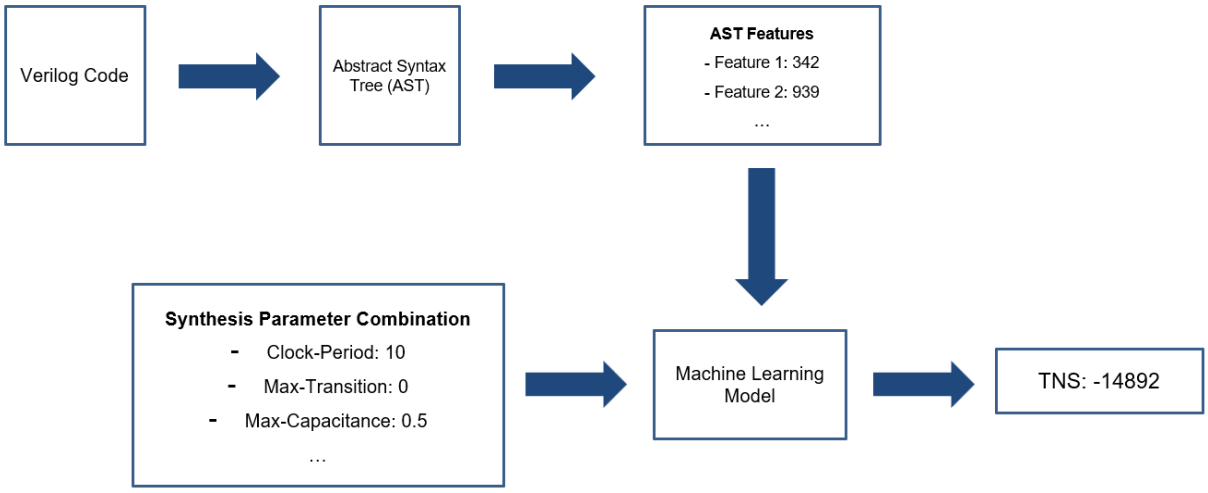


*Figure 2: TNS prediction using Sengupta's ML-based process*

One bottleneck in Sengupta's work is the preliminary time necessary to generate training data for the machine learning model. According to Sengupta's approach, the following steps must be performed to generate the training data:

1. Choose *m* synthesis recipes.
    a. A synthesis recipe is defined as the set of input parameters for the synthesis tool, excluding the clock period, to guide the process of netlist generation.
2. Choose *n* benchmark Verilog designs expressed in a hardware description language (HDL) such as Verilog.
3. Choose a clock period for each design.
4. For each design, run logic synthesis and placement with each of the *m* synthesis recipes and the design's chosen clock period.

This procedure involves performing $n * m$ synthesis and placement runs. Since each run takes hours to perform, the procedure—even when distributed across multiple machines—can take many days to complete, assuming that *n* and *m* are relatively large. In [1], 45 designs ($n = 45$) and 416 synthesis recipes ($m = 416$) were used for training the model. Consequently, $n * m = 45 * 416 = 18720$ synthesis and placement runs were performed to generate all the training data. The procedure, distributed across multiple machines, took approximately a week to complete. There is a great need for a technique that reduces the number of synthesis and placement runs needed to generate training data without negatively impacting the accuracy of TNS prediction. Facilitating the collection of training data will increase initial adoptions of Sengupta's machine learning framework by users and organizations to predict the TNS of their designs without wasting valuable RTL developer time running logic synthesis and placement after each design iteration.

### 1.2 Inspiration

After discovering a remarkable relationship between the TNS values and synthesis recipes across the various designs in our data set of 29 different designs, a path for reducing synthesis and placement runs was revealed. Logic synthesis and placement were run on each design with 429 different synthesis recipes to collect the designs' true TNS values. The clock period parameter was kept constant within each design depending on the requirements of that design. Figures 3 and 4 display the TNS values for 6 different designs – *vga_lcd*, *des3_area*, *gfx*, *systemcdes*, *sasc*, and *des3_perf*. Figure 3 displays the TNS values in a scatter plot format. The x-axis of each scatter plot corresponds to unique synthesis recipes, and the y-axis corresponds to the resulting TNS values. Although the range of the TNS values differs across designs, the shape of the distribution of the TNS values remains similar. More specifically, for each synthesis recipe

*r*, the point corresponding to *r* appears to remain in the same location relative to all the other data points on each scatter plot. Figure 4 displays the TNS values in a histogram format. Like the scatterplots, the histograms show that the distribution of TNS values across designs is similar. The other 20 not-pictured designs within the data set show similar TNS scatterplots and histograms.

Since the TNS values are similarly distributed for different designs, two distinct approaches for reducing the number of logic synthesis and placement runs are proposed. One method to generate training data with fewer runs is to scale and shift the TNS values of a carefully selected "base design" to generate the TNS values of other designs. The other method of constructing training data with minimal runs is the following. First, the synthesis recipes are divided among the designs, after which synthesis and placement are run to determine the TNS values. Then, these TNS values, after being scaled and shifted, are stored in a "base vector", which is used to generate the TNS values of other designs.

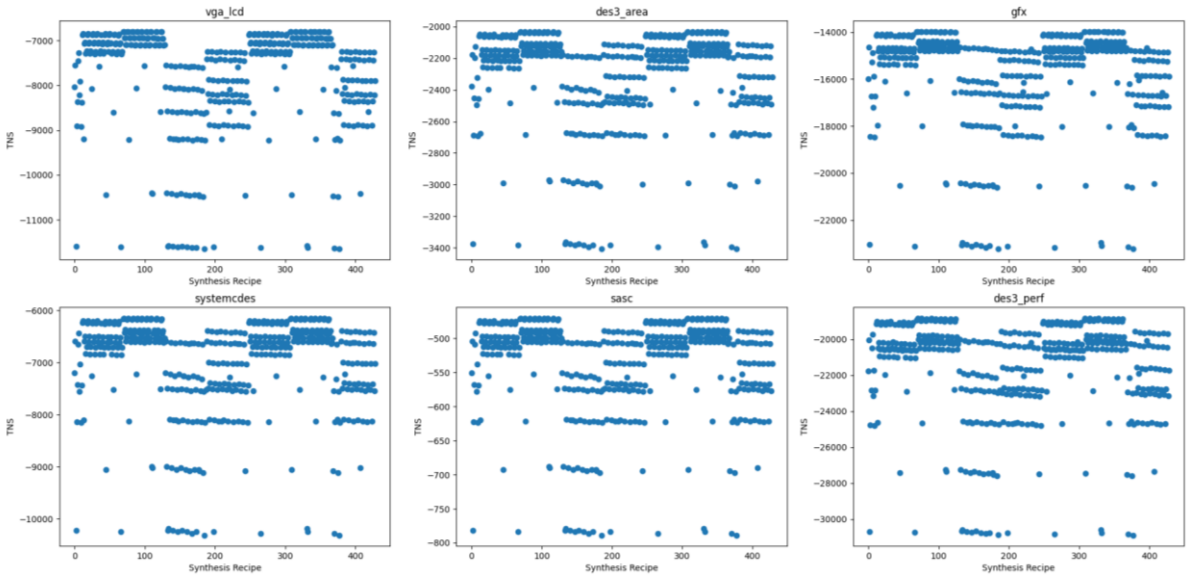


*Figure 3: TNS scatter plots for 429 synthesis recipes and 6 designs*

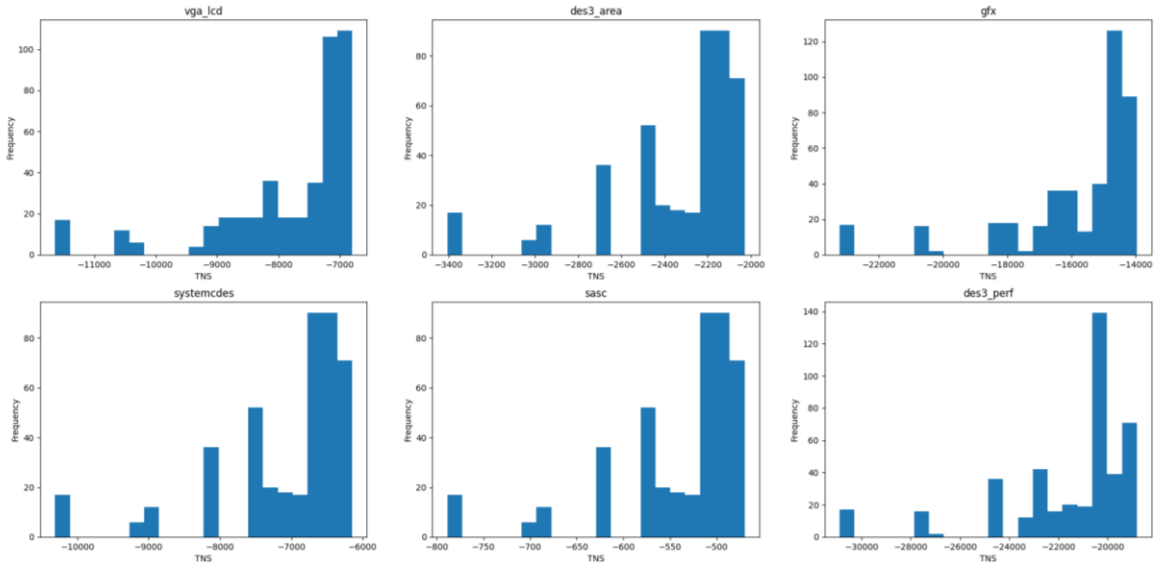


*Figure 4: TNS histograms for 429 synthesis recipes and 6 designs*

While the scatter plots and histograms of TNS showed an interesting relationship between different designs, negating and applying the natural log function, $\ln(-TNS)$, to each value produced particularly illuminating distributions. Since the TNS values have drastically different ranges for different designs, the natural log function was applied to the TNS values with the theory that the transformed TNS distributions would have similar ranges. TNS is negated before applying the natural log function because the TNS values in the data set are strictly negative and the function is defined only for positive values.

Figures 5 and 6 show the transformed TNS values for 6 different designs. Figure 5 displays the transformed TNS values in a scatter plot format. The x-axis of each scatter plot corresponds to unique synthesis recipes, and the y-axis corresponds to transformed TNS values. Like the TNS values in Figure 3, the transformed TNS values are distributed similarly across designs. Unlike the figure, however, the transformed TNS values have identical ranges across designs. For example, the transformed TNS values of *des3_area* vary approximately from 7.65 to 8.15, which is a range of 0.5. The transformed TNS values of *gfx* vary approximately from

9.55 to 10.5, which is also a range of 0.5. The histograms in Figure 6 show this observation as well. Since the transformation procedure "normalizes" the TNS values, this process will be referred to as normalization. After normalization, the TNS distributions of each design are now a simple translation away from each other.

We hypothesize that this normalization procedure can be used to scale the TNS values for the two previously mentioned approaches. The two approaches can now be described with more precision. In the first approach, a representative "base design" is chosen based on distribution sampling, then its full TNS distribution is normalized using the $\ln(-TNS)$ function. For every other design, this normalized distribution is translated by a computed magnitude, and the inverse operations are applied to convert back to raw TNS, generating the values of that design. In the second approach, the synthesis recipes are divided among the designs, after which synthesis and placement are run to collect the TNS values. These TNS values, after being normalized and shifted, are pieced together to form an aggregated "base vector". For every design, this base vector is translated by a computed magnitude, and the inverse operations are applied to convert back to raw TNS, generating the values of that design. Using these approaches has the potential to substantially reduce the number of synthesis and placement runs needed to train machine learning models to predict TNS.

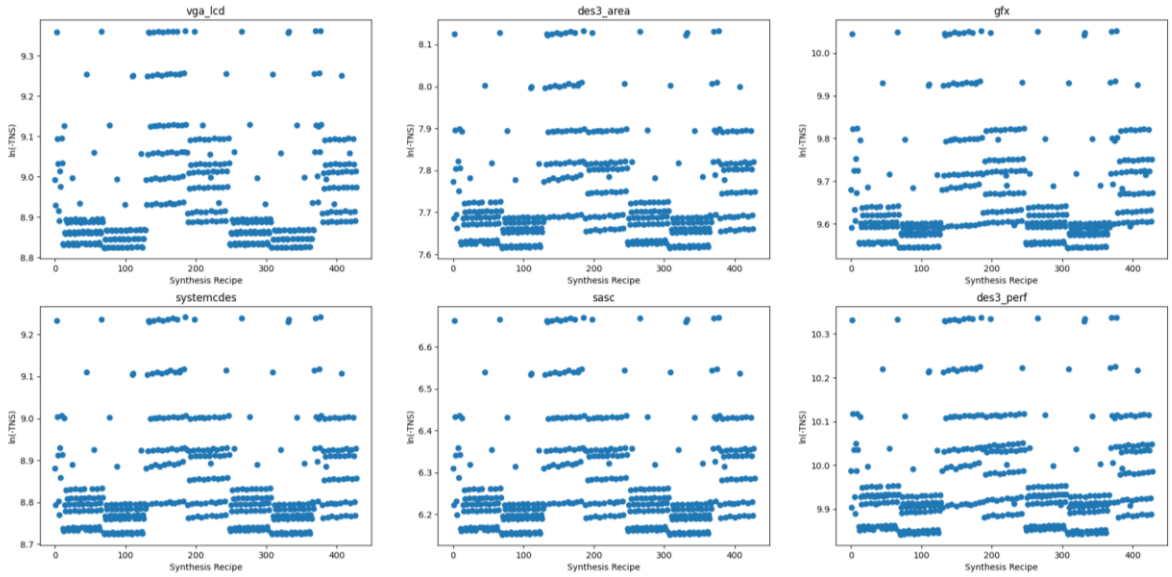


*Figure 5: Normalized scatter plots for 429 synthesis recipes and 6 designs*

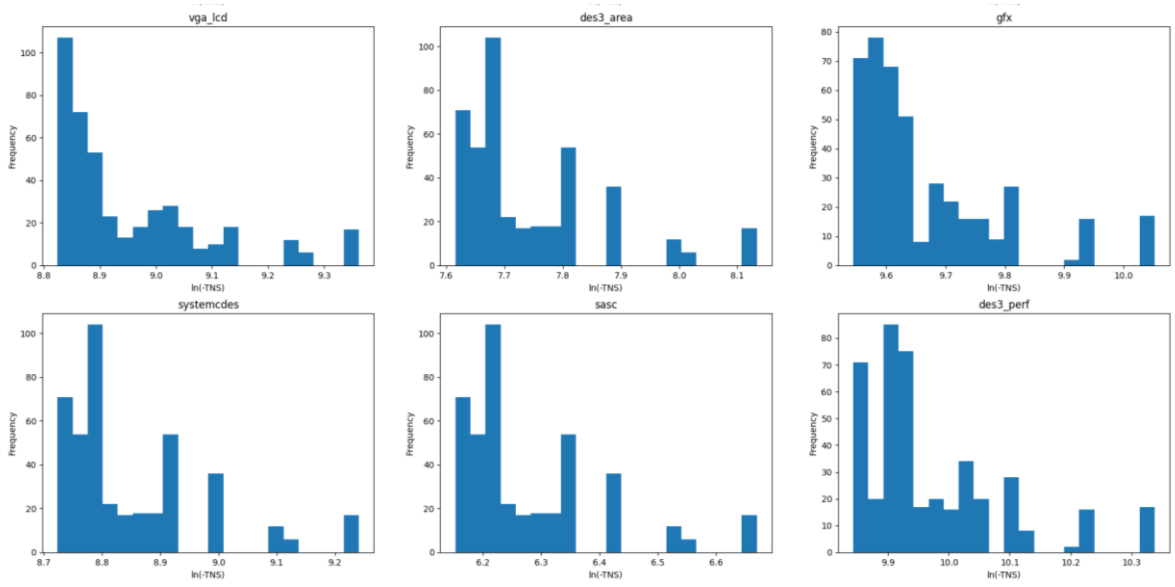


*Figure 6: Normalized TNS histograms for 429 synthesis recipes and 6 designs*

To ensure that these patterns are not a result of the designs themselves being similar to one another, the source of the designs and the code itself were further analyzed. The benchmark designs used in this research come from the *International Workshop on Logic and Synthesis* conference that took place in June 2005 [2]. The RTL code was written by various open-source

developers and compiled into a collection for public use during and after the conference. The chips were designed independently by experts in the field of logic synthesis, and each chip performs a unique and identifiable computing task. The chips also contain unique numbers of sequential and combinational cells. Further analysis revealed that variable names and coding styles in the Verilog code were unique, making it unlikely that the designs are copies of one another or chained together to make new designs in the same set. Finally, to generate the initial target data for our research and get the "true" TNS values for each design, logic synthesis was run using the Synopsys Design Compiler, and placement was run using the Cadence Innovus Implementation System.

### 1.3 Overview

In this paper, two different approaches to reducing synthesis and placement runs are described, both of which were inspired by the findings discussed in the prior section. Both approaches assume that the true TNS values, which are determined by running the synthesis and placement tools, are strictly negative (non-zero) for the training designs paired with synthesis recipes. The first approach is labeled as the "base design approach," while the second approach is labeled as the "aggregated approach". The number of RTL designs in the data set is denoted as *n*, and the number of unique logic synthesis recipes used for training is denoted as *m*. The base design approach requires $5n + m - 5$ total synthesis and placement runs, while the aggregated approach requires only $n + m - 1$ total synthesis and placement runs. The basis of these expressions is described in the next section. The approach requiring $n * m$ synthesis and placement runs is labeled as the "traditional approach" and serves as the control group for the experiment. The base design and aggregated approaches are tested to find whether they can reduce the number of runs without compromising the accuracy of the machine-learning models.

To analyze the results, the efficiency of training data generation and the accuracy of the resulting machine learning models for the traditional, base design, and aggregated approaches are compared against one another.

# 2. METHODS

## 2.1 Traditional Approach

The accuracy of machine learning models trained using the traditional approach serves as a control group for the experiment. The machine learning models chosen for testing are random forest and XGBoost because they are models used by [1], and the *Scikit-learn* library's implementation of these models [3] is readily available. The configuration of the models is shown below:

- Random Forest - 100 trees, max depth of 10, random state of 42
- XGBoost - 80 trees, max depth of 7, random state of 42

The above configurations are the same configurations used within [1], excluding the random state, which is not provided.

The data set used in the experiment contains 29 different designs. A mockup of this data set is shown in Table 1.

*Table 1: Example data set*

| | AST Feature Values | Clock Period | Synthesis Parameter Recipe | TNS Value |
|---|---|---|---|---|
| **Design 1** | [1, 2, 3, 4, 5] | 1 | [0 0 0 0] | -20 |
| **Design 1** | [1, 2, 3, 4, 5] | 1 | [0 0 0 1] | -25 |
| **Design 1** | [1, 2, 3, 4, 5] | 1 | [0 0 1 0] | -30 |
| **Design 1** | [1, 2, 3, 4, 5] | 1 | [0 0 1 1] | -35 |
| **Design 1** | [1, 2, 3, 4, 5] | 1 | [0 1 0 0] | -40 |

| | | | | |
|---|---|---|---|---|
| **Design 2** | [2, 3, 4, 5, 6] | 2 | [0 0 0 0] | -200 |
| **Design 2** | [2, 3, 4, 5, 6] | 2 | [0 0 0 1] | -250 |
| **Design 2** | [2, 3, 4, 5, 6] | 2 | [0 0 1 0] | -300 |
| **Design 2** | [2, 3, 4, 5, 6] | 2 | [0 0 1 1] | -350 |
| **Design 2** | [2, 3, 4, 5, 6] | 2 | [0 1 0 0] | -400 |

For each of the 29 designs, the design was converted into an AST using Pyverilog [4], then certain features were extracted from the AST using the algorithm designed by Sengupta. This is shown in the *AST Feature Values* column. A clock period was also chosen for each design based on individual requirements, shown in the *Clock Period* column. Logic synthesis and placement were run on each design for 429 different synthesis recipes. These recipes are shown in the *Synthesis Parameter Recipe* column. The designs were randomly partitioned into a training and test set. The training set contains 70% of the designs, while the test set contains 30% of the designs. In other words, twenty randomly chosen designs are labeled for training, while the remaining nine are labeled for testing.

### *2.1.1 Results*

The correlation coefficient R-squared (defined in Equation 1) measures the accuracy of the machine learning models on the training and test sets.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y}_i)^2} = \frac{MSE(mean\ model)\ -\ MSE(new\ model)}{MSE(mean\ model)} \quad (1)$$

The correlation coefficient is interpreted as the percentage decrease in the mean squared error from using the mean model (a model that only returns the average result) to using the new model (our machine learning model). R-squared will have a low value of 0 if the new model has

a mean squared error equivalent to that of the mean model, meaning that the new model performs no better than picking the average TNS value in the data set each time. R-squared will have a high value of 1 if the new model has a mean squared error of 0, meaning that the model perfectly predicts every TNS value in the data set. R-squared will have a value of less than 0 if the new model has a larger mean squared error than the mean model.

*Table 2: R-squared of machine learning models trained using the traditional approach*

| | Training R-squared | Testing R-squared |
|---|---|---|
| **Random Forest** | 0.999 | 0.844 |
| **XGBoost** | 0.999 | 0.912 |

Table 2 shows the results of training machine learning models using the traditional approach. Both random forest and XGBoost have a very high training R-squared of 0.999. The testing R-squared of random-forest is relatively high at 0.844, and the training R-squared of XGBoost is much higher at 0.912. Because the training R-squared is very high and the testing R-squared is much lower for both models, the models could be overfitting to the training data set.

## 2.2 Base Design Approach

The first approach to reducing synthesis and placement runs is the base design approach. At a high level, the base design approach performs the following steps. To begin, five synthesis recipes are chosen at random, and logic synthesis and placement are performed using those recipes on all *n* designs. The TNS values generated by these 5 "common" synthesis and placement runs across all *n* designs are analyzed to determine an ideal "base design" whose distribution matches most of the designs in the data set. On this base design, the remaining $m - 5$ synthesis and placement runs are performed to generate the base design's complete set of TNS

values. This completed distribution is used along with one of the 5 common synthesis and placement runs to generate the TNS values of all other designs by normalizing and shifting the base design's TNS distribution. The base design approach requires $5n + m - 5$ total synthesis and placement runs, and machine learning models trained using this approach perform with similar accuracy to those using the traditional approach. Algorithm 1 describes the procedure for selecting a base design and generating the complete training data set from a set of *n* Verilog designs and *m* unique synthesis recipes using the base design approach.

**Algorithm 1: Base Design Approach**

**Input:**
***designs*** *(a list of n designs)*
***synth_recipes*** *(a list of m synthesis recipes)*
**Output:**
***gen_tns*** *(a table containing the generated TNS values)*

***true_tns*** ← *an empty table with the rows* ***designs*** *and columns* ***synth_recipes***
***rrecipes*** ← *5 randomly chosen synthesis recipes from* ***synth_recipes***

**for** ***d*** *in* ***designs***
    **for** ***r*** *in* ***rrecipes***
        *run logic synthesis and placement on* ***d*** *using* ***r***
        ***true_tns[d][r]*** ← *the run's TNS value*
    **end**
**end**

***std_devs*** ← *an empty list*
**for** ***d*** *in* ***designs***
    *append the standard deviation of ln(-****true_tns[d][rrecipes]****) to* ***std_devs***
**end**

*create a plot of* ***std_devs*** *vs* ***designs***
// on the plot, most designs lie along a horizontal line denoted as the "majority line"
***bdesign*** ← *a design on the majority line with a small synthesis and placement runtime*

**for** ***r*** *in* ***synth_recipes*** *excluding* ***rrecipes***
    *run logic synthesis and placement on* ***bdesign*** *using* ***r***
    ***true_tns[bdesign][r]*** ← *the run's TNS value*
**end**

***rrecipe*** ← *a randomly chosen synthesis recipe from* ***rrecipes***

```
26  gen_tns ← a copy of true_tns
27
28  for d in designs excluding bdesign
29  |   run logic synthesis and placement on d using rrecipe
30  |   true_tns[d][rrecipe] ← the run's TNS value
31  |   shift ← ln(-true_tns[d][rrecipe]) - ln(-true_tns[bdesign][rrecipe])
32  |   gen_tns[d][synth_recipes]) ← -exp(ln(-true_tns[bdesign][synth_recipes]) + shift)
33  end
34
35  return gen_tns
```

### *2.2.1 Choosing the base design*

The majority of the 29 designs in the data set had identical TNS distributions. This can be observed by examining the similarity between the plots of the following designs in Figures 3 and 4: *des3_area*, *systemcdes*, and *sasc*. Several other designs had similar, but slightly different distributions. Examples of these can be visualized by examining the plots of the following designs in the same figures: *vga_lcd*, *gfx*, and *des3_perf*. In the base design algorithm, a "base design" is chosen by finding a design with a TNS distribution that most closely matches the TNS distributions of the majority of the designs. From Figures 3 and 4, *des3_area*, *systemcdes*, and *sasc* are base design candidates. The base design's TNS values are scaled and shifted to produce the TNS values of other designs using the normalization technique described earlier. Because the base design has a distribution shared by the majority of the designs, it can accurately approximate the TNS distribution of other designs. Moreover, since even outlier distributions, such as *gfx*'s distribution, are similar but not identical to the majority distribution, the base design's distribution is still an adequate approximation.

The base design approach chooses a base design in lines 1 through 18 of Algorithm 1. The algorithm starts by randomly choosing 5 synthesis recipes, which are used to run logic synthesis and placement on all the designs. The 5 TNS values for each of the designs are

normalized and their standard deviation is plotted on a scatterplot. The majority of the designs lie along a horizontal line denoted as the “majority line”. The base design is determined by choosing a design on the majority line with a small synthesis and placement runtime.

To choose a base design without running $n * m$ synthesis and placement runs and then comparing the TNS distributions of designs, the base design approach makes use of the following fact. Given two different designs and a few random synthesis recipes, if the normalized TNS values for the designs using those recipes have identical standard deviations, then the two designs likely have similar normalized TNS distributions, meaning that their raw TNS distributions are also similar. In practice, the standard deviation of 5 random synthesis recipes is sufficient to determine whether the TNS distributions of two designs are similar. Since a majority of the designs have the same TNS distribution, the majority of TNS standard deviations of designs for 5 common synthesis recipes are the same. This produces a horizontal line, known as the “majority line,” on the scatterplot of each design and the standard deviation of its 5 TNS values as shown in Figure 7. Any of the designs with points along the majority line are adequate candidates for the base design since a design lying on the majority line implies that the design has the majority distribution. However, since $m - 5$ synthesis and placement runs are performed on the base design later in the procedure, choosing a model on which logic synthesis and placement run quickly helps to save time in the long run. For example, if given the choice between a small design and a large design as the base design, the smaller design is chosen because running synthesis and placement $m - 5$ times will finish sooner.

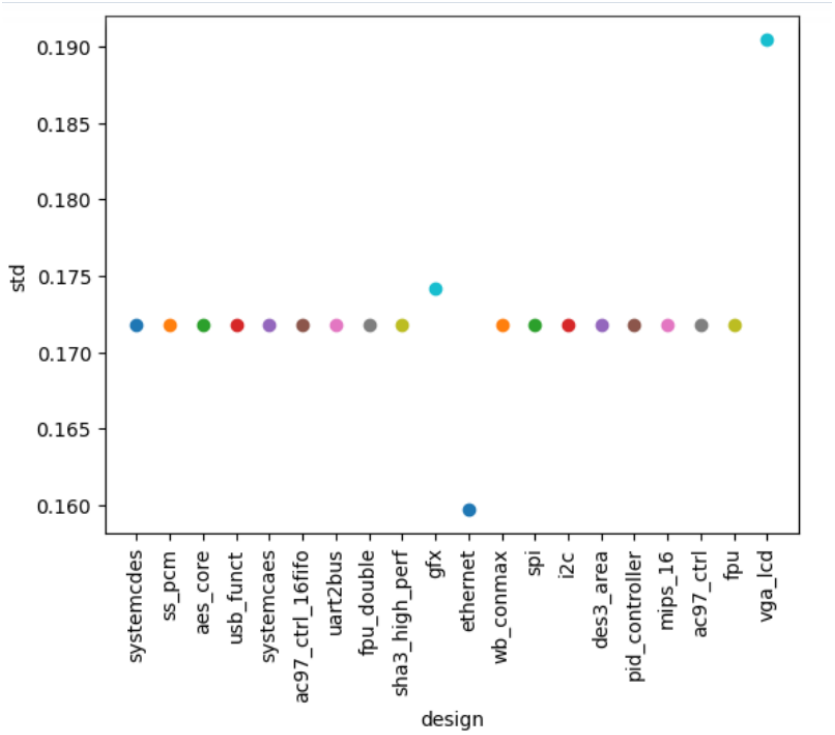


*Figure 7: Standard deviation of normalized TNS values generated from 5 random synthesis recipes*

In the data set of 29 designs, the same training set of 20 designs used to test the traditional approach was used to test the base design approach. 5 synthesis recipes were randomly chosen, logic synthesis and placement were run with those recipes on each design, and the resulting TNS values were normalized and recorded. Figure 7 graphs the standard deviation of these normalized distributions on the y-axis, and the corresponding design on the x-axis. The majority line on which most of the data points lie can be seen at 0.172, and only three designs deviate from this majority line. Any of the points on the majority line can be selected as the base design, but for this experiment, the design corresponding to the first point on the majority line, *systemcdes*, was chosen. This design's TNS distribution can be observed in Figures 3 and 4.

### *2.2.2 Using the base design*

Once the base design has been chosen, logic synthesis and placement must be performed on the design multiple times to collect its TNS values. These TNS values, along with the TNS

values of one of the 5 runs already performed on all the other designs, can be used to generate the TNS values of all other designs once normalized.

The base design approach generates TNS values by making use of the base design in lines 20 through 35 of Algorithm 1. The algorithm starts by performing synthesis and placement runs on the base design using all the synthesis recipes (excluding the 5 common synthesis recipes). Then, one random recipe is chosen from the common synthesis recipes. For each design (excluding the base design), the TNS value for the random recipe is collected and normalized. This normalized TNS value is subtracted by the normalized TNS value of the base design for the random recipe and stored in the *shift* symbol. The *shift* symbol is used to translate the normalized TNS values of the base design, which are then transformed using the negative exponential function generate the design's TNS values.

Since the base design's TNS values are distributed identically to the majority of the designs, the base design approach collects them by performing $m - 5$ synthesis and placement runs so they can be used to generate the TNS values of other designs. Since normalized TNS values of different designs have the same range, the normalized TNS values only need to be translated by a certain magnitude to generate the normalized values of other designs. The algorithm determines this magnitude for each design by performing the following steps. First, one synthesis recipe is randomly chosen from the 5 common synthesis recipes. Then, the magnitude for a design is calculated by taking the difference between the design's normalized TNS value and the base design's normalized TNS value at the chosen synthesis recipe. After applying the appropriate translations to all the data points, the normalized TNS values for that design are generated. To get the raw TNS from the normalized $\ln(-TNS)$ value, the inverse operations are applied for each data point, as shown in Equation 2.

$$TNS = -e^{\ln(-TNS)} \quad (2)$$

Using our base design *systemcdes*, for which TNS values corresponding to 429 different synthesis recipes had already been collected, the TNS values of the other 19 designs within the training data set were generated. One of the 5 common synthesis recipes was chosen randomly and used to translate the normalized values of *systemcdes* to generate the normalized values of the other designs. These normalized values were transformed back to raw TNS values using Equation 2.

### *2.2.3 Results*

*Table 3: R-squared of machine learning models trained using the base design approach*

| | **Training R-squared** | **Testing R-squared** |
|---|---|---|
| **Random Forest** | 0.999 | 0.805 |
| **XGBoost** | 0.999 | 0.897 |

After generating the TNS values of 19 designs using the TNS values of the base design, *systemcdes*, the training data set contained 429 TNS values for each of the 20 designs. To understand how the accuracy of machine learning models would be affected by training with the base design approach as opposed to the traditional approach, machine learning models were trained on the generated data set, and their accuracy was measured using the R-squared correlation coefficient defined in Equation 1. The same data that was used in the traditional approach's testing data set was used for the testing data set. Table 3 shows the results of training machine learning models using the base design approach. Both random forest and XGBoost

have very high training R-squared of 0.999. The testing R-squared of random forest is relatively high at 0.805, while the testing R-squared of XGBoost is significantly higher at 0.897. There is a small decrease in R-squared from the traditional approach to the base design approach for both random forest and XGBoost, but both decreases are less than 0.05.

## 2.3 Aggregated Approach

The second approach to reducing synthesis and placement runs is the aggregated approach. At a high level, the aggregated approach performs the following steps. To begin, the *m* synthesis recipes are randomly distributed to the *n* designs. Logic synthesis and placement are then performed on each design with all the synthesis recipes assigned to it. Then, one "common" synthesis recipe is chosen at random, and logic synthesis is performed using that recipe on all designs. For each design, the normalized TNS values for the assigned synthesis recipes are shifted by the normalized TNS value for the common synthesis recipe, which are then stored in an aggregated "base vector." Finally, the base vector is shifted by the TNS values generated using the common synthesis recipe to generate the TNS values of all designs. The aggregated approach requires $n + m - 1$ synthesis and placement runs. When training machine learning models using the aggregated approach, one of them has similar accuracy to those trained using the traditional approach. Algorithm 2 describes the procedure for generating a base vector and the complete training data set from a set of *n* Verilog designs and *m* unique synthesis recipes using the aggregated approach.

**Algorithm 2: Aggregated Approach**

**Input:**

***designs*** *(a list of n designs)*

***synth_recipes*** *(a list of m synthesis recipes)*

**Output:**

***gen_tns*** *(a table containing the generated TNS values)*

```
true_tns ← an empty table with the rows designs and columns synth_recipes
shuffled_recipes ← a randomly shuffled copy of synth_recipes
recipe_chunks ← a list of n equal chunks of shuffled_recipes
chunk_dict ← an empty dictionary

for i in 1 to n
    chunk_dict[designs[i]] ← recipe_chunks[i]
end

for d in designs
    chunk ← chunk_dict[d]
    for r in chunk
        run logic synthesis and placement on d using r
        true_tns[d][r] ← the run's TNS value
    end
end

rrecipe ← a randomly chosen synthesis recipe from synth_recipes
base_vector ← an empty one-dimensional table with columns synth_recipes

for d in designs
    chunk ← chunk_dict[d]
    if rrecipe is not in chunk
        run logic synthesis and placement on d using rrecipe
        true_tns[d][rrecipe] ← the run's TNS value
    end
    base_vector[chunk] ← ln(-true_tns[d][chunk]) – ln(-true_tns[d][rrecipe])
end

gen_tns ← an empty table with the rows designs and columns synth_recipes

for d in designs
    gen_tns[d][synth_recipes]) ← -exp(base_vector + ln(-true_tns[d][rrecipe]))
end

return gen_tns
```

### *2.3.1 Generating the base vector*

In the base design algorithm, $5n$ initial synthesis runs were required to choose a base design. Since the aggregated approach does not need to choose a base design, these initial synthesis and placement runs are not needed. Instead of choosing a base design, the aggregated

approach uses a “base vector”. The base vector has a distribution that is similar to the normalized distributions of the majority of the designs, meaning that it can accurately approximate the TNS distributions of other designs.

The aggregated approach constructs the base vector in lines 1 through 28 of Algorithm 2. The algorithm starts by randomly shuffling the synthesis recipes and splitting them into *n* equal chunks. Each chunk is assigned to a design by initializing the dictionary *chunk_dict*. For each design, synthesis and placement are run using the synthesis recipes of the design’s corresponding chunk. A random recipe is then chosen from the synthesis recipes. For each design, synthesis and placement are run using the random recipe. The normalized TNS value of this recipe is subtracted from the normalized TNS values for the chunk of the design. The result is saved to the base vector. Once this procedure has been completed for each of the designs, the base vector is created.

The reasoning for why the base vector’s distribution is similar to the normalized distributions is as follows. After synthesis and placement are run for each design on its corresponding chunk, the resulting values, when normalized and stored together, do not form a distribution similar to the majority normalized TNS distribution. This is because the normalized TNS distributions of different designs are translations away from one another. The translation between normalized distributions of different designs can be removed by subtracting each design’s normalized distributions by the design’s normalized TNS value for a common synthesis recipe. Consequently, the aggregated approach shifts the resulting values for each design by a normalized TNS value determined by the running synthesis and placement using a common synthesis recipe.

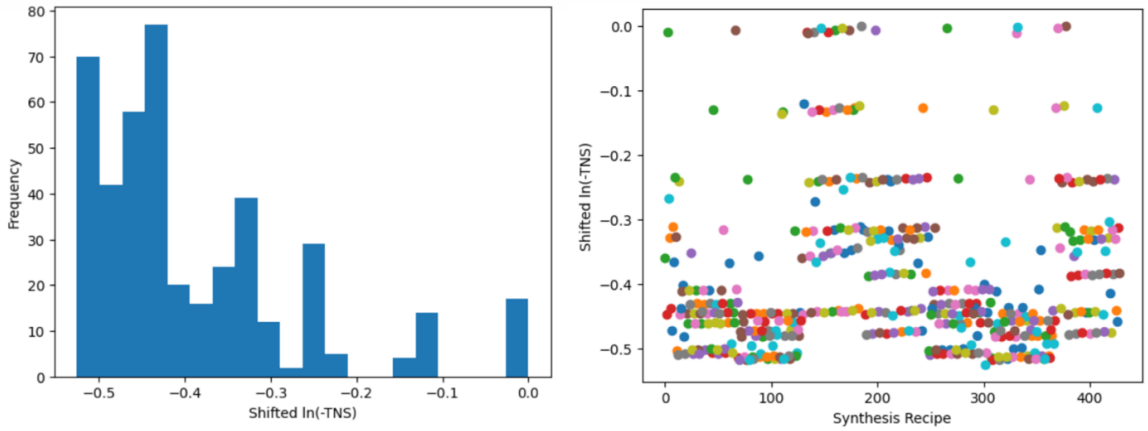


*Figure 8: Shifted and normalized TNS histogram and scatterplot of the base vector*

The same training set of 20 designs that was used to test the traditional and base design approaches was used to test the aggregated approach. The 429 synthesis recipes were distributed evenly among the 20 designs, logic synthesis and placement were run for each recipe on the design it was assigned to, and the resulting TNS values, after being shifted and normalized, were recorded in the base vector. Figure 8 illustrates the values in the base vector. The scatter plot shows the shifted and normalized TNS values for different synthesis recipe. Each color corresponds to a chunk of synthesis recipes assigned to a particular design. The histogram shows the base vector's overall distribution. The similarity between the base vector's distribution and the majority normalized TNS distribution can be seen by comparing the base vector's histogram with Figure 5.

### *2.3.2 Using the base vector*

Once the base vector has been constructed, one synthesis recipe is chosen, and the corresponding TNS value is collected for all *n* designs using logic synthesis and placement. The aggregated approach generates the TNS values of all designs in lines 30 through 36 of Algorithm 2.

The shifted, normalized TNS values of the base vector are distributed similarly to the majority of the designs and can be used to generate the normalized TNS values of other designs. Since the base vector's values were determined by subtracting the normalized TNS value from a common synthesis recipe, a normalized TNS value from the common synthesis recipe must be added back to generate the normalized TNS values of any design. The negative exponential function is then applied to generate the actual TNS values.

### *2.3.3 Results*

*Table 4: R-squared of machine learning models trained using the aggregated approach*

| | **Training R-squared** | **Testing R-squared** |
|---|---|---|
| **Random Forest** | 0.999 | 0.775 |
| **XGBoost** | 0.999 | 0.922 |

After generating the TNS values of 19 designs using the TNS values of the base vector, the training data set contained 429 TNS values for each of the 20 designs. To understand how the accuracy of machine learning models would be affected by training with the aggregated approach as opposed to the traditional approach, machine learning models were trained on the generated data set and their accuracy was measured using the R-squared correlation coefficient defined in Equation 1. The same data that was used in the traditional approach's testing data set was used for the testing data set. Table 4 shows the results of training machine learning models using the aggregated approach. Both random forest and XGBoost have very high training R-squared of 0.999. The testing R-squared of random forest is relatively low at 0.775, while the testing R-squared of XGBoost is much higher at 0.922. There is a large decrease in the R-squared from the traditional approach to the aggregated approach for random-forest.

Surprisingly, there is a small increase in the R-squared from the traditional approach to the aggregated approach for XGBoost.

# 3. EVALUATION

## 3.1 Efficiency

To better visualize the reduction in synthesis and placement runs that the novel approaches produce when generating training data for the machine learning models, Figure 9 displays how the total number of required synthesis and placement runs changes with either the number of benchmark designs or the number of unique synthesis recipes in the data set. The formulas are shown below, where *n* represents the number of benchmark designs and *m* represents the number of unique synthesis recipes.

- Traditional Approach $n * m$
- Base Design Approach $5n + m - 5$
- Aggregated Approach $n + m - 1$

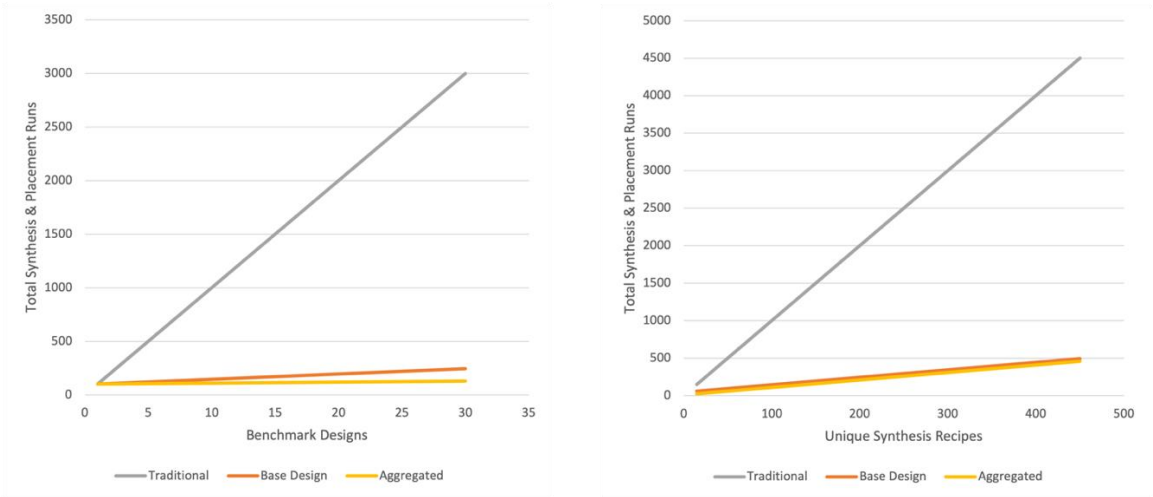


*Figure 9: Relationship between # of benchmark designs/synthesis recipes and total synthesis & placement runs*

When either the number of unique synthesis recipes or the number of benchmark designs is kept constant (at arbitrary values of $m = 100$ and $n = 10$ respectively) and the other is varied, the effect on the total required synthesis and placement runs is the same. The base design

approach requires slightly more synthesis and placement runs than the aggregated approach. On the other hand, the traditional approach requires an order of magnitude more synthesis and placement runs for a data set of the same size, which translates to significantly more time running synthesis and placement. As the size of the data set increases, the difference in time spent generating the training data set for the traditional and novel approaches increases quadratically.

The next section evaluates whether the novel approaches compromise machine learning model accuracy to achieve this time reduction.

## 3.2 Accuracy

Random forest and XGBoost machine learning models were trained on the training data generated by each approach to evaluate their performance. The outcomes of each approach (traditional, base design, and aggregated) on the accuracy of the machine learning models are summarized in Figure 10.

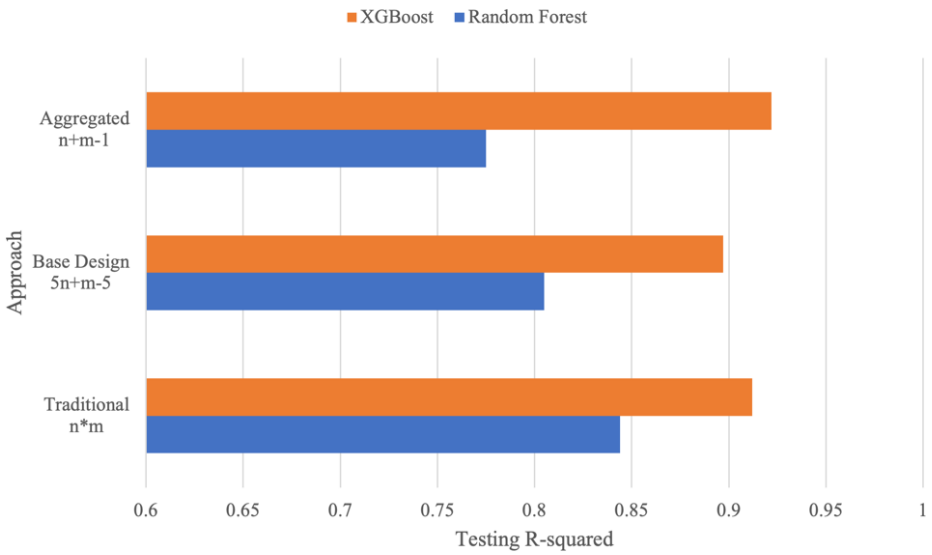


*Figure 10: Accuracy of ML models after using different approaches to generate their training data*

The effect of each approach differs for random forest and XGBoost machine learning models. For random forest models, the traditional approach to generating the training data—where logic synthesis and placement were run for all $n * m$ combinations of synthesis recipe and Verilog design—had the best performance, with an R-squared of 0.844. As the number of synthesis runs to generate training data decreased, the accuracy of random forest models also decreased. The base design approach, which required $5n + m - 5$ total logic synthesis and placement runs, resulted in a random forest R-squared of 0.805, while the aggregated approach required $n + m - 1$ total logic synthesis and placement runs and resulted in a random forest R-squared of 0.775. Both are a significant decrease in accuracy from the traditional approach. This pattern lines up with intuition because when the training set contains more "artificial" data, the accuracy of the resulting predictions is diminished.

XGBoost models displayed more unpredictable behavior with the varying number of synthesis runs. The base design approach resulted in the lowest XGBoost R-squared at 0.897, while the traditional approach had an XGBoost R-squared of 0.912. Unexpectedly, the XGBoost model performed with higher accuracy when trained on data generated from the aggregated approach as opposed to data generated from the traditional approach. This outcome is surprising because, as mentioned previously, the aggregated approach training data contains the highest percentage of "artificial" data out of the three approaches tested. Theoretically, this should have resulted in the lowest accuracy since the training data set looked the most different from the test data set.

Overall, the base design and aggregated approaches did not produce great results for the random forest machine learning models. The base design and aggregated approaches reduced the number of synthesis and placement runs necessary to generate training data compared to the

traditional approach, but this reduction caused a tradeoff with the random forest model's prediction accuracy and compromised its performance. The XGBoost machine learning models, on the other hand, displayed promising results with the two novel approaches to generating training data. The base design approach caused a minimal decrease in the model's prediction accuracy, but the aggregated approach caused a slight increase in prediction accuracy over the traditional approach.

# 4. CONCLUSION

While conducting this research, our team had the opportunity to learn more about the hardware design process and the challenges it faces in the industry. We experienced firsthand how time-consuming the logic synthesis and placement & routing processes are and how we could apply machine learning and data analysis techniques to streamline them. Receiving instant feedback on RTL code using machine learning enables chip designers to work several times more efficiently because longer sprints of development without running the full synthesis and PnR processes are possible within a design iteration.

This research focused on simplifying the setup phase of the machine learning models by reducing the number of synthesis and placement runs necessary to generate training data from a set of benchmark RTL designs. Previous work required running synthesis on every single design/synthesis recipe combination to find the correct TNS, which quickly grew to tens of thousands of synthesis and placement runs for just a handful of designs in the data set. By taking advantage of similarities in the TNS distributions of each design across the set of common synthesis recipes, we were able to reduce the number of synthesis and placement runs from $O(nm)$ in the traditional approach to $O(n + m)$ with near-perfect accuracy of prediction in most cases. The two methods we analyzed were the "base design" and "aggregated" approaches. In the base design approach, 5 synthesis recipes are chosen to run on all the designs in the data set except one chosen base design, with which all *m* synthesis recipes are run to compute the full TNS distribution. In the aggregated approach, each synthesis recipe is randomly assigned to one of the benchmark designs to run synthesis on, generating a base vector that serves as the reference distribution used to make predictions.

The progress made in this research will assist users of Sengupta's AST parsing algorithm [1] with training their machine-learning models by streamlining the process of adding new designs to their data set. Adding new designs to the data set sooner results in more comprehensive and accurate models, which are beneficial for providing RTL designers with precise metrics to converge on design requirements in fewer design iterations.

## 4.1 RTL_QoR_Predictor

The *RTL_QoR_Predictor* is a product of our research that allows any stakeholders or interested parties in academia or industry to use the AST parsing engine (designed by Sengupta [1] and implemented by us) and some sample pre-trained machine learning models on their own Verilog RTL code to see it in action, as well as provide feedback on bugs or other potential improvements.

The *RTL_QoR_Predictor* was written in Python and packaged into a UNIX executable with Pyinstaller for public use.

# REFERENCES


[1] 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 (ICCAD ’22)*, San Diego, California, December 2022, pp. 1–9. Available: https://doi.org/10.1145/3508352.3549375.

[2] C. Albrecht, "IWLS 2005 Benchmarks," in *Proc. IWLS '05*, June 2005.

[3] F. Pedregosa et al., “Scikit-learn: Machine Learning in Python”, *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[4] S. Takamaeda-Yamazaki, “Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL”, in *Applied Reconfigurable Computing*, 2015, vol. 9040, pp. 451–460. Available: http://doi.org/10.1007/978-3-319-16214-0_42.