Read your Circuit: Leveraging Word Embedding to Guide Logic Optimization

Walter Lau Neto[1], Matheus Trevisan Moreira[2], Luca Amaru[3], Cunxi Yu[1], Pierre-Emmanuel Gaillardon[1]
[1]Department of Computer and Electrical Engineering, University of Utah, Salt Lake City, Utah, USA
[2]Chronos Tech, San Diego, California, USA
[3]Synopsys Inc., Design Group, Sunnyvale, California, USA

ABSTRACT

To tackle the involved complexity, *Electronic Design Automation* (EDA) tools are broken in well-defined steps, each operating at different abstraction levels. Higher levels of abstraction shorten the flow run-time while sacrificing correlation with the physical circuit implementation. Bridging this gap between Logic Synthesis tool and *Physical Design* (PnR) tools is key to improve *Quality of Results* (QoR), while possibly shorting the time-to-market. To address this problem, in this work, we formalize logic paths as sentences, with the gates being a bag of words. Thus, we show how word embedding can be leveraged to represent generic paths and predict if a given path is likely to be critical post-PnR. We present the effectiveness of our approach, with accuracy over than 90% for our test-cases. Finally, we give a step further and introduce an intelligent and non-intrusive flow that uses this information to guide optimization. Our flow presents up to 15.53% area delay product (ADP) and 18.56% power delay product (PDP), compared to a standard flow.

KEYWORDS

Timing Closure; Electronic Design Automation; Word Embedding; Machine Learning

ACM Reference Format:
Walter Lau Neto[1], Matheus Trevisan Moreira[2], Luca Amaru[3], Cunxi Yu[1], Pierre-Emmanuel Gaillardon[1]. 2021. Read your Circuit: Leveraging Word Embedding to Guide Logic Optimization. In *26th Asia and South Pacific Design Automation Conference (ASPDAC '21), January 18–21, 2021, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3394885.3431560

1 INTRODUCTION

As the technology evolved, *Integrated Circuits* (ICs) had their number of transistor doubling every two years, according to Moore's Law [14]. To keep peace with such a rocket in integration, *Electronic Design Automation* (EDA) tools play a crucial role in automating the design of ICs, composed of billions of transistors. To make it possible, the EDA flow is decoupled in well-defined steps, as follows [12]: (i) High-level synthesis, (ii) logic synthesis, and (iii) physical design. To make it practical and affordable in run-time, every step operates at a different abstraction level, omitting some information handled by the following steps. For instance, logic synthesis abstracts some of the physical information while performing logic optimization, which leads to a miss-correlation with the physical layout generated at the end of the flow, impacting the quality of the final IC [21]. On the other hand, physical design has detailed information about all the aspects of the final IC. However, it does not support aggressive logic optimizations. Thus, common issues that are only found after physical design cause many iterations looping back to the front-end flow to run more aggressive optimizations and achieve design closure.

In this context, there is a great need to bridge the gap between steps, improving their correlation, and leading to (i) designs with better *Power-Performance-Area* (PPA), and (ii) achieving faster design closure. This paper explores the recent advances in *Machine Learning* (ML) to bridge logic synthesis and physical design. More specifically, we propose a comprehensive method to embed generic logic paths into a low-dimensional space of representation, being so capable of not only predict post *Place-and-Route* (PnR) critical paths before technology mapping, but also making use of such information to *guide* the optimization algorithms throughout the entire flow. We showcase the benefits of such an approach over an *Advanced Encryption Standard* (AES) core, where we improve PPA metrics over a standard-commercial flow. While other academic works apply ML for either logic synthesis or physical designs, for the best of our knowledge, that is the first work that systemically guides optimization algorithms, since the first steps of logic synthesis, to improve post-PnR predicted critical paths.

The main contributions of this paper are as follows:**1)** we propose a **path embedding** method that allows a CNN to take (i) a logic path in terms of *generic* gates, and (ii) the target cycle time (CT) the circuit should operate, and classify the path as being critical or not. **2)** we evaluate the power of the proposed embedding method by building a simple CNN model that reaches **over 95% of accuracy** in fairly complex, and never seen, designs. **3)** we then **close the loop**, and introduce a flow that makes use of the proposed embedding method along with the CNN model. The proposed flow is *non-intrusive* and can be easily employed in any conventional EDA flow. It guides the logic synthesis front-end to optimize the critical paths since the generic synthesis. **4)** we demonstrate improvements of up to 15.53% area delay product (ADP) and 18.56% power delay product (PDP) when applying the proposed flow, compared to a standard

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPDAC '21, January 18–21, 2021, Tokyo, Japan
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-7999-1/21/01...$15.00
https://doi.org/10.1145/3394885.3431560

commercial flow. **5)** we make available all the Python source code along with the data-sets publicly available[1].

2 BACKGROUND

Convolutional Neural Networks (CNNs) have been successfully applied in many domains where the data can be represented in a grid-like fashion [10]. Logic paths are a good fit for CNNs, as it can be seen as a 1-Dimensional grid. Furthermore, the convolution operation is usually meaningful when there is a sense of spatial correlation. In a logic path, the context where the logic gates appear has an important role. As we are interested in predicting critical paths, our CNN works as a binary-classifier, and predicts if a path is likely to be critical or not, given a target timing constraint. Common metrics to evaluate a binary-classifier are: (i) its ($precision(p)$), (ii) its $recall(r)$, and (iii) its $F1$ score. *Precision* indicates the total of true positives divided by the sum of true positive and false positives, i.e., the proportion of paths predicted to be critical and are critical. The *recall* component can be calculated by the total of true positives divided by the sum of true positive and false negatives, i.e., the proportion of the paths that are critical and were correctly predicted. Finally, the $F1score$ is the harmonic mean of precision and recall.

Logic Synthesis and Physical Design: Logic synthesis is divided into two major steps: technology-independent and technology-dependent. In this work, we handle networks at the technology-independent level. Hence, the tool represents the circuit as a *Directed Acyclic Graph* (DAG), where each vertex corresponds to a Boolean primitive. Common DAGs for technology-independent manipulation include *And-Inverter Graph* (AIG) [20], or *Majority-Inverter Graph* (MIG) [1]. In our case, we adopted a commercial tool that used as basic functions: *not*, *and2*, *nand2*, *or2*, *nor2*, *complex2*, and *flip-flops*. After generic optimization, the circuit goes through technology-mapping, and then physical design, which delivers the final circuit implementation. Technology mapping and physical design are out of this work scope, and we refer the reader to [4, 12] for further details. Here, we note that: (i) the input graph coming from the logic synthesis tool plays a key role in the physical design flow, and (ii) physical design tools have limited optimization capabilities, being usually limited to gate sizing and buffering. Thus, if the designer cannot attain the design constraints, he needs to re-run the circuit throughout the entire flow, manually tuning the many steps to optimize the paths that failed to achieve timing closure. Unfortunately, running a circuit throughout the EDA flow is a time-consuming task; hence, optimizing the right paths in the early stages is highly beneficial.

Fig. 1 shows a motivational example of how the different levels of abstractions on the tool make predicting critical paths in a generic netlist challenging. The blue dots represent a path, where the X coordinate is the path delay post-generic synthesis, and the Y coordinate is the same path delay post-PnR. The horizontal green line represents the target cycle time we aim the circuit to meet. On the other hand, the horizontal violet line delimits points within 10% of the target cycle time, *i.e.,* sensitive paths that may violate the timing constraint. The red line is the identity function, and if the path delay were not changing from generic synthesis to physical design, all the paths (blue dots) would be on the top of the line. In the Figure, it is possible to note that paths in the range of 85 ps to 140 ps vary a lot, making it hard to predict if a path will become critical or not.

[1]https://github.com/mtmoreira/pathAnalyzer

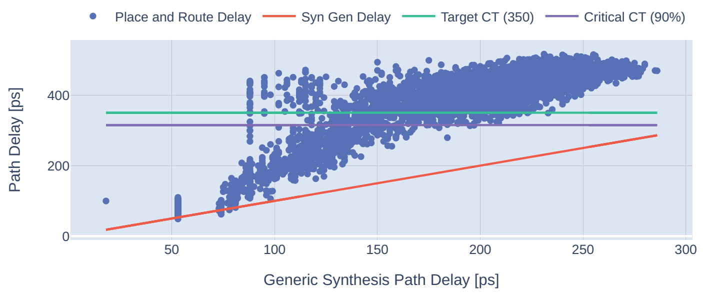

Figure 1: The X axis represents the generic synthesis delay, whereas the Y axis denotes the picture legend delay for the different lines. This plot presents that relying on generic synthesis delay is not reliable as logic path delays change significantly post-PnR .

3 RELATED WORKS

This section starts by discussing proposed Embedding approaches related to our work. We then present related works applying ML in the EDA context, for both Logic Synthesis and Physical Design.

3.1 Embedding Approaches

Circuits do not have a straight representation in a vector space, which is necessary to make it applicable in ML, as it enables mathematical operations ubiquitous to ML. Thus, to embed logical paths in a low-dimensional space, we borrow concepts from *Natural Language Processing* (NLP). More specifically, we note that a logic path is a chain of logic gates, in the same way that a phrase is a chain of words. Word embedding aims to represent words in a multi-dimensional space onto a continuous vector space, composed of real numbers [13]. Words with a similar meaning in context are embedded close, i.e., are encoded with similar vectors. The embedding values are trainable and capture the relationship between words. We note that in the same way in which words have a different meaning depending upon the context they appear in, the delay of the logic path also depends on the path structure and on the context the gates appear.

3.2 Machine Learning in EDA

On the Logic Synthesis side, *Haaswijk et al.* [6] apply *Reinforcement Learning* during technology-independent optimization. The optimization problem is formulated as a single-player game composed of a set of states and moves. States correspond to the logic network, whereas moves correspond to transformations in the network. Results show the capability of the model to find the optimal size implementation of all 3-input functions. *Yu et al.* [21] proposes a CNN-based approach to generate ASIC design-specific logic-synthesis flows autonomously in the ABC environment [20], showing that different flows may lead to circuit delay and area variations of up to 40% and 90%, respectively. The proposed CNN model is capable of autonomously providing flows with good and bad *Quality-of-Results* (QoR). Recently, in [15] the authors propose a technology-independent optimization framework that uses different technology-independent representations. The idea is to leverage optimization opportunities enabled by different logic representations. Thus, the authors propose to represent sub-networks with a Karnaugh-map image. Sub-networks may be optimized with either AIGs or MIGs. The authors present QoR improvements compared to the state-of-the-art open-source framework ABC [20].

According to [8] prominent application of ML in the physical design includes removing unnecessary design margins through correlation mechanisms and achieving fast design convergence by predicting outcomes of the physical design flow. [9] proposed a machine learning approach to estimate wire delay/slew and avoid calling a timing analysis tool multiple times during the physical design gate sizing optimization. In [7] the authors present a deep learning approach capable of correlating timing information between different tools. The idea is that the designer may use more than one timing analysis tool to check if the design is over(under)constrained, and tools from different vendors may diverge in the timing results. In [3], the authors propose an ML approach to predict detailed-routing *Design Rule Check* (DRC) violation, and then estimates detailed placement to reduce these violations. [22] propose a model based on *Generative Adversarial Networks* (GANs) to forecast routing congestion after design placement for FPGAs. The model input is a post-placement image and outputs a congestion heat map predicted by the model. [11] casts analytical placement as a neural network training problem. They propose DREAMPlace, which is accelerated by GPU and achieves a 30x speedup in global placement without QoR degradation.

Even though machine learning has been applied in both domains independently, there are still missing approaches that try to fill the gap between logic synthesis and physical design, which is the primary motivation of this work. In this sense, this work aims to predict the outcomes of PnR early in the logic synthesis flow, removing unnecessary margins and guiding optimization. In this sense, our approach differs from *Static Timing Analysis* (STA) as it does not rely on high-level models to estimate timing but on how the structure of the path is going to change through the flow to predict how critical a path is. Also, it is not called many times in the optimization loop, such as STA.

4 PROPOSED APPROACH FOR LOGIC PATH EMBEDDING

This section starts by introducing the proposed path embedding approach. We discuss the feature selection, data labeling, and the embedding itself. Then, we give one step further and present a flow to close the loop and make practical use of our classifier. We discuss the proposed flow and show how it fits any conventional EDA flow without being intrusive.

4.1 Feature Selection and Data Labeling

To define the features to be used, we focused on the information available in a generic netlist. Therefore, we have set up different strategies that take into account different metrics available after generic synthesis, namely: (i) the gate logic function; (ii) the input transition direction; (iii) the cell fanout; (iv) the cell load; (v) the cell delay; and (vi) the target cycle time. We combined these metrics in different ways to test the classifier accuracy. Since we do not aim to predict the exact path delay but rather classify if a path is likely to be critical post PnR, we can focus only on the most important features. Our experiments show that three features are *sufficient* to achieve good precision. Hence, we embed the gates with the *design target cycle time*, the *gate logic function*, and the *gate load*. The gate logic function plays an important role, as it defines the transistors' arrangement to implement a given Boolean function, *i.e.,* transistors sizing and the stack of transistors, which dominates the gate delay [17]. The gate load, on the other hand, gives a flavor about the path surrounding. It enables the model to estimate the logic being driven by a given gate and accounts for the tool's wire load model. The slew rate in the input and the pair of input-output being excited have a much more reduced effect in the delay of a gate. Also, the direction of the input transition has shown to have a low impact on our model.

Data Labeling: To label our data, we have two main steps. After generic synthesis, for each pair of start-point/end-point, we collect all the gates composing the path. In the sequence, we proceed throughout the complete flow without any extra constraints. At the end of the entire flow, we can check the logic path's actual delay, post-physical design, with more accurate wire-models and with a delay precisely computed by the *Static Timing Analysis* (STA) tool. Note that, even though the paths structure changes during the implementation flow, the start-point and end-point have their names preserved. Therefore, we can match a generic path by the start-point/end-point name after PnR and capture how its structure and delay changed. If this delay satisfies a criticality definition, we label this path as critical (1). Otherwise, we mark it as non-critical (0). In our setup, we define that any logic path that at the end of the implementation flow has its delay larger or equal to 90% of the target cycle time will be considered a critical path.

4.2 Path Embedding

Although the target cycle time and output load are numerical values, the gate function is not. Thus, we need to represent the logic function as a value and embed it along with the other features to represent a path. That is the first contribution of this paper. As for the cell name, we tried two different encodings. The first, using the *embed layer* of tensor flow, and the second on with a custom embedding where the values try to approximate the logic cell logical effort [19]. Our method has shown to be more stable, with constant accuracy. We believe that it also represents the gates' logical effort, which, along with the other features, makes our embedding robust. Hence, we map cell functions with values monotonically growing according to the complexity of the function. We define that a ***not*** has a value of **1.0**, ***nand2*** and ***nor2*** have a value of **2.0**, ***and2*** and ***or2*** have a value of **3.0**, ***complex2*** has a value of **4.0** and a ***flop*** has a value of **5.0**. That allows us to represent logic gates with low-dimensional numerical features, giving us a stable classification accuracy. To guide the reader through our embedding, let's consider the circuit extracted from a generic synthesis in Fig. 2.

As the figure shows, this circuit has three start-points (g0, g1, and g2) and two end-points (g10 and g11), and a total of 7 valid structural paths. Table 1 shows the delays and output loads for each gate. These values were collected with the STA tool available in the adopted flow after generic synthesis. The paths delay along with the gates composing each path is presented in Table 2.

First, we collect the critical path for each pair of start-point/end-point. This information gives us a good representation of the circuit

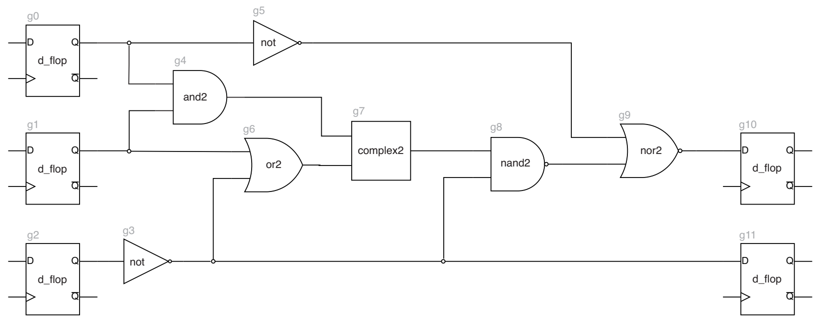

Figure 2: Example circuit fragment with generic gates *not*, *nand2*, *nor2*, *and2*, *or2*, *complex2*, and *d_flop*.

Table 1: Output loads and delays of gates of example circuit fragment in Fig. 2 in load and time units.

Gate	Function	Output Load	Delay (*ps*)
g0	d_flop	5	35
g1	d_flop	10	40
g2	d_flop	3	30
g3	not	4	7
g4	and2	5	20
g5	not	2	5
g6	or2	6	22
g7	complex2	4	27
g8	nand2	3	15
g9	nor2	4	17
g10	d_flop	4	32
g11	d_flop	5	35

Table 2: Logic paths of example circuit fragment in Fig. 2. ID is the index of the path, SP is its startpoint, EP is its endpoint, Gates is a list of the gates in it and Delay is the total path delay in time units. Endpoints are removed from gate lists, as they do not add to path delay.

ID	SP	EP	Gates	Delay (*ps*)
0	g0	g10	g0, g5, g9	57
1	g0	g10	g0, g4, g7, g8, g9	114
2	g1	g10	g1, g4, g7, g8, g9	119
3	g1	g10	g1, g6, g7, g8, g9	121
4	g2	g10	g2, g3, g6, g7, g8, g9	118
5	g2	g10	g2, g3, g8, g9	69
6	g2	g11	g2, g3	37

logic structure, with its worst paths at generic synthesis time. For example, for the pair g0 and g10, path 1 is the critical, as it has a delay of 110, while path 0 has a delay of 55.

Next, to embed these paths as numerical features, we define them as a vector of gates, where each gate is a vector of floating-point numbers encoding the gate features. To prevent biasing the model, we normalize the output load and the target cycle time.

In simple terms, a path is modeled as a matrix with L rows and 3 columns. In this matrix, L is the path's size (or the number of gates in the path). Each row of the matrix represents a logic gate, and the columns are the target cycle time, the name of the logic gate, and its output load, respectively. The target cycle time is repeated for every gate to model its relationship with the total budget for the path's delay. Because all features must be of the same size, the definition of the size L of a path is given by the largest structural path in the data-set. We pad paths with the number of logic gates smaller than L with a vector of 0s, eliminating the effect of cells in that position in the overall model. To enable homogeneously

0.125	3.0	0.3
0.125	1.0	0.4
0.125	2.0	0.6
0.125	4.0	0.4
0.125	2.0	0.3
0.125	2.0	0.4

(a) a

0.125	5.0	0.3
0.125	1.0	0.4
0.0	0.0	0.0
0.0	0.0	0.0
0.0	0.0	0.0
0.0	0.0	0.0

(b) b

Figure 3: Embedding example for paths 4 (a) and 6 (b), from the circuit fragment in Fig. 2.

data-set, we borrowed NLP concepts and added the padding in the tail of our matrix.

In our example, the largest path has 6 gates. Let us assume that this circuit has a target cycle time of 125, and the gates name modeled by the approximation of their logic effort, as previously discussed. Using the load information from Table 1, we can build the features for path 4, showed in Fig. 3(a), and path 6, showed in Fig. 3(b).

Note that we ignored setup time in flip-flops in this example for the sake of simplicity. However, this information is automatically accounted for by EDA tools when exporting logic path information. Also, although the example only presents paths with start-points and end-points as flops, our strategy captures paths from *in2out*, *in2reg*, *reg2reg*, and *reg2out*.

4.3 Architecture and Hyper-parameters

The CNN input layer comprises L neurons, with L being the path length available on the data-set. Each neuron has a width of three to represent the gate features. The input layer is then connected to a 3x3 convolutional filter, which aims to relate each gate delay and output load with the specified target time and its neighboring gates. The number of filters in the convolutional layer is 32, and the filter stride is 1, as we do not exceed the path length. Finally, the convolutional layer is connected to a single output neuron. All the activation functions are sigmoids. The number of convolutional filters and the CNN architecture was defined by performing multiple rounds of training and testing. We apply Adam optimizer and a cross-entropy loss function for training.

4.4 Binary Classifier Results and Discussion

4.4.1 ***Methodology***. To train our model, we choose an open-source RISC-V core[2]. We decide to train our model over a RISC-V processor mainly for two reasons. First, it is an open-source hardware *Instruction Set Architecture* (ISA), with high adoption in the

[2]https://github.com/cliffordwolf/picorv32.git

industry. Second, it is a well-balanced circuit, *i.e.,* it has significant portions of arithmetic and control logic.

Dataset Generation: to generate our data-set, we run a standard EDA flow and set the target cycle time ranging from 400 ps to 1000 ps, in steps of 100 ps. That is to ensure we collect paths under different scenarios, *i.e.,* with cycle times the tool cannot meet, with tight cycle times, and with relaxed cycle times. These constraints have a great impact on how the tool structures the logic paths. Our data-set samples the *worst path* for each-endpoint, so that we have all the *in2out*, *in2reg*, *reg2reg*, and *reg2out* pairs of paths, collected after the generic synthesis. The labels are created according to the path delays collected after the PnR flow. Fig. 4 shows the distribution of path size during synthesis generic, as box plots on the y axis, for each criticality, on the x-axis. Criticality is defined as explained in section 4.1. This plot presents two interesting insights: (i) for each critical point, we have a good variety of paths with distinct lengths; and (ii) we can see that paths from basically any length may become critical. It also shows our data-set is well balanced, as we have a wide range of path lengths for the different critical points.

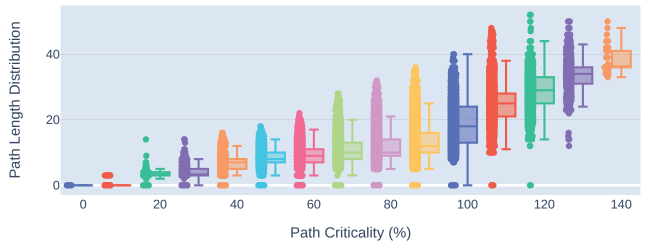

Figure 4: Criticality distribution per path length: The x axis presents the criticality of a given path, whereas the y axis shows the path length. It is possible to see there is a good distribution of path lengths for the different critical regions.

4.4.2 Binary Classifier Discussion. We train our model for 10 epochs over the paths extracted from the RISC-V core. First, we shuffle the data-set, and randomly select 70% of the paths for training, with the remaining 30% of paths for validation. As for the model testing, we extract the *in2out*, *in2reg*, *reg2reg*, and *reg2out* pairs of paths for two designs. The first is an AES cryptography core, a widely adopted module in circuit design, with fair portions of arithmetic and control logic. The second is the exponential module in the *NVIDIA Deep Learning Accelerator* (NVDLA) [16]. Both circuits operate in a frequency range similar to the RISC-V core and allow us to verify the generality of the method.

Classifier Accuracy: first, let us present the classifier results while classifying the paths of an AES targeting a cycle time of 350 ps. Our binary classifier model presents an $F1$ score of 0.952, where the components $precision(p)$ is equal to 0.996, and the component $recall(r)$ is 0.911. Table 3 presents the classifier confusion matrix. As for the NVDLA module, we achieve an F1 of 93.46%, with p = 96.56%, and r = 90.50%. Even though we present results for a single cycle time, these metrics remain similar while applying a different timing constraint.

Note that we have such a large amount of positive samples because we are pushing the tool to its limit. If the tool can easily meet the constraints, then the proposed method is not required, as no further optimizations are needed to achieve design closure.

	Predicted Positive	Predicted Negative
Actual Positive	5,570	539
Actual Negative	17	1,192

Table 3: AES Confusion Matrix.

5 CLOSING THE LOOP WITH EASY OPT

In this section, we introduce an intelligent flow that makes use of the information given by the classifier, forcing the entire flow to perform aggressive optimization in the paths predicted to be critical by our model. Fig. 5 presents an overview of our proposed system. As the diagram shows, the first step is to perform a generic synthesis and generate a report of the critical paths between each start-point and end-point pair and their target cycle times. That is done with a custom TCL script. This information is the input of our Path Embedding tool, which transforms information about these paths into numerical features. The generated features are the input to a CNN-based binary classifier model that will decide what paths will become critical to the circuit being implemented. With this information, we generate a constraint file indicating to the tool which paths are critical and should be more heavily optimized. This constraint file is then read, and the system continues, as in a traditional EDA flow. Note that the extra step added in the flow has a negligible impact over the flow run-time. Path extraction can be performed quickly, and path embedding, as well as the model inference, are instantaneously computed.

The constraint file feeds back to guide the flow from technology mapping until its end and aims to automatize a manual when design-closure is not achieved. We have chosen to constraint the paths with the command "*set_max_delay*", with the flags *-from* the start point and *-to* the end-point. The max delay value is set to 0.

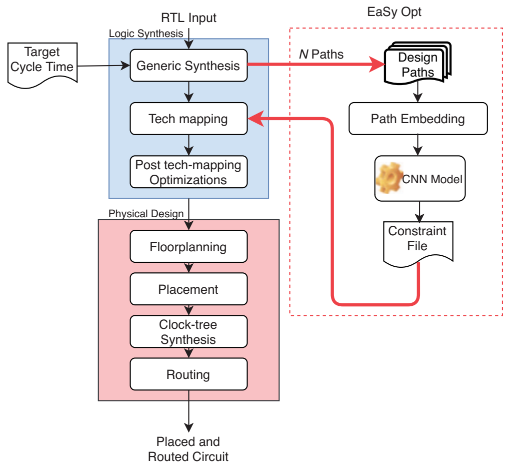

Figure 5: *EaSy Opt* System Overview. Standard flow follows the top-down dark arrows. Our system is interfaced with the standard flow as indicated by the red arrows.

5.1 Improving QoR by Predicting and Constraining Critical Paths

This section presents the QoR improvements achieved while applying the proposed EaSyOpt flow to design the AES core while

Table 4: Post PnR results comparison while designing an AES core for different cycle times (CT). Area is given in μm^2, achieved CT is given in ns, and power dissipation in mW. ADP stands for *Area Delay Product*, whereas PDP stands for *Power Delay Product*

Target CT	Standard Flow					EaSy Opt					ADP	EDP
(ps)	Area (μm^2)	Achieved CT (ns)	Power (mW)	ADP	PDP	Area (μm^2)	Achieved CT (ns)	Power (mW)	ADP	PDP	Improvement	Improvement
225	26,968.80	479.39	10.42	1.29E+07	5.00E+03	27,839.16	448.36	10.51	1.25E+07	4.71E+03	3.45%	5.66%
275	26,354.10	489.38	8.66	1.29E+07	4.24E+03	27,894.45	435.64	8.62	1.22E+07	3.76E+03	5.78%	11.39%
325	26,108.69	505.35	7.24	1.32E+07	3.66E+03	26,768.18	443.38	7.05	1.19E+07	3.13E+03	10.05%	14.57%
350	26,000.02	513.21	6.65	1.33E+07	3.41E+03	25,997.89	433.60	6.41	1.13E+07	2.78E+03	15.53%	18.56%

using an industrial EDA standard tool-chain. We show standard metrics for evaluating circuit design QoR, such as (i) circuit area; (ii) achieved CT; and (iii) power dissipation. To select the cycle time constraint used in both the reference and propose flow, we run the standard flow using a wide range of different cycle times to understand which CTs the tool can meet with ease. Then, we select some points below this cycle time to show the benefits of our method.

Table 4 depicts the achieved results when applying the *EaSy Opt* flow. Red data indicates a worse result, whereas blue denotes an improvement. It is possible to notice that the proposed flow achieves better performance than the standard flow for all the cases and better power dissipation in most cases. As one would expect, both improvements come at the price of area degradation. That is expected as area and performance are opposite goals, and most of the techniques that lead to performance improvements come at an area cost. Thus, *Area Delay Product* (ADP) and *Power Delay Product* (PDP) capture this trade-off. [18]. In this sense, we improve ADP up to 15.53% and PDP up to 18.56% compared to the standard flow. Note that achieving such improvements in the figures of merit is significant and non-trivial. Any gains over a state-of-the-art industrial flow are considered significant [2, 5].

6 CONCLUSION

This paper investigates how to represent generic logic paths as tensors to predict their post-PnR criticality. We show that post-generic synthesis features are limited, but sufficient. Then, we propose an analogy of cells as words and paths as phrases to embed logic paths into tensors. The proposed embedding achieves over than 90% of accuracy in two complex designs. To show the implications of our approach, we give one step further and introduce the EaSyOpt, a portable and non-intrusive flow that uses our CNN structure to guide the whole EDA flow. EaSyOpt is highly portable among different vendors and presents ADP and PDP improvements of up to 15.53% and 18.56%, respectively. It also improves performance for all the cases, which is the main focus of our flow. By doing so, we expect to achieve design closure faster, likely shortening the time-to-market. Future works include investigating a minimalist data-set generation to achieve generalization and further exploration of different optimization commands as well as other techniques to classify critical paths.

7 ACKNOWLEDGMENTS

This research was partially supported by the Defense Advanced Research Projects Agency (DARPA - FA8650-18-2-7849) and National Science Foundation (NSF-CCF-2008144). The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory and Defense Advanced Research Projects Agency or the U.S. Government.

REFERENCES

[1] Luca Amaru, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2015. Majority-inverter graph: A new paradigm for logic optimization. *In IEEE TCAD* (2015), 806–819.
[2] L. Amarú, P. Vuillod, J. Luo, and J. Olson. 2017. Logic optimization and synthesis: Trends and directions in industry. In *DATE, 2017.* 1303–1305.
[3] Wei-Ting J. Chan, Pei-Hsin Ho, Andrew B. Kahng, and Prashant Saxena. 2017. Routability Optimization for Industrial Designs at Sub-14nm Process Nodes Using Machine Learning. In *ISPD 2017.* 15–21.
[4] Satrajit Chatterjee, Alan Mishchenko, Robert K Brayton, Xinning Wang, and Timothy Kam. 2006. Reducing structural bias in technology mapping. *In IEEE TCAD* (2006), 2894–2903.
[5] W. Haaswijk, L. G. Amarù, P. Vuillod, J. Luo, M. Soeken, and G. De Micheli. 2018. Integrated ESOP Refactoring for Industrial Designs. In *2018 25th ICECS.* 369–372.
[6] W. Haaswijk, E. Collins, B. Seguin, M. Soeken, F. Kaplan, S. Süsstrunk, and G. De Micheli. 2018. Deep Learning for Logic Optimization Algorithms. In *2018 ISCAS.* 1–4.
[7] Seung-Soo Han, Andrew B Kahng, Siddhartha Nath, and Ashok S Vydyanathan. [n.d.]. A deep learning methodology to proliferate golden signoff timing. In *2014 IEEE DATE.* 1–6.
[8] Andrew B Kahng. 2018. Machine learning applications in physical design: recent results and directions. In *Proc. of the 2018 ISPD.* 68–73.
[9] A. B. Kahng, S. Kang, H. Lee, S. Nath, and J. Wadhwani. 2013. Learning-based approximation of interconnect delay and slew in signoff timing tools. In *2013 ACM/IEEE SLIP.* 1–8.
[10] Yann LeCun, Yoshua Bengio, et al. 1995. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks* 3361, 10 (1995), 1995.
[11] Yibo Lin, Shounak Dhar, Wuxi Li, Haoxing Ren, Brucek Khailany, and David Z Pan. 2019. DREAMPlace: Deep learning toolkit-enabled GPU acceleration for modern VLSI placement. In *Proc. of the 56th Annual DAC 2019.* 1–6.
[12] Giovanni De Micheli. 1994. *Synthesis and optimization of digital circuits.* McGraw-Hill Higher Education.
[13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).
[14] Gordon E Moore et al. 1965. Cramming more components onto integrated circuits. *Electronics Magazine* 8 (1965).
[15] W. L. Neto, M. Austin, S. Temple, L. Amaru, X. Tang, and P. Gaillardon. 2019. LSOracle: a Logic Synthesis Framework Driven by Artificial Intelligence: Invited Paper. In *2019 IEEE ICCAD.* 1–6.
[16] NVIDIA. 2020. NVIDIA Deep Learning Accelerator (NVDLA). https://github.com/nvdla
[17] V. N. Possani, V. Callegaro, A. I. Reis, R. P. Ribas, F. de Souza Marques, and L. S. da Rosa. 2016. Graph-Based Transistor Network Generation Method for Supergate Design. *In IEEE TVLSI* (2016), 692–705.
[18] Jan M Rabaey, Anantha P Chandrakasan, and Borivoje Nikolić. 2003. *Digital integrated circuits: a design perspective.* Vol. 7. Pearson Education Upper Saddle River, NJ.
[19] Ivan Sutherland, Robert F Sproull, Bob Sproull, and David Harris. 1999. *Logical effort: designing fast CMOS circuits.* Morgan Kaufmann.
[20] Berkeley Logic Synthesis and Verification Group. 2020. ABC: A System for Sequential Synthesis and Verification. https://github.com/berkeley-abc/abc.git
[21] Cunxi Yu, Houping Xiao, and Giovanni De Micheli. 2018. Developing synthesis flows without human knowledge. In *Proc. of the 55th Annual DAC.* ACM, 50.
[22] Cunxi Yu and Zhiru Zhang. 2019. Painting on Placement: Forecasting Routing Congestion Using Conditional Generative Adversarial Nets. In *Proc. of the 56th DAC 2019.* 6.