Concurrent Sign-off Timing Optimization via Deep Steiner Points Refinement

Siting Liu$^{1,2}$†, Ziyi Wang$^1$, Fangzhou Liu$^1$, Yibo Lin$^2$, Bei Yu$^1$, Martin Wong$^2$
$^1$Chinese University of Hong Kong  $^2$Peking University

Abstract—Timing closure is crucial across the circuit design flow. Since obtaining sign-off performance needs a time-consuming routing flow, all the previous early-stage timing optimization works only focus on improving early timing metrics, e.g., rough timing estimation using linear RC model or pre-routing path-length. However, there is no consistency guarantee between early-stage metrics and sign-off timing performance. To enable explicit early-stage optimization on the sign-off timing metrics, we propose a novel timing optimization framework, TSteiner. This paper demonstrates the ability of the learning framework to perform robust and efficient timing optimization in the early stage with comprehensive and convincing experimental results on real-world designs.

I. INTRODUCTION

Modern design flow requires iteratively invoking design stages like placement and routing (PnR) for sign-off timing closure, which is considerably time-consuming. As a result, there arises an urgent demand for improving sign-off timing performance directly in the early stages to reduce the expensive PnR iterations.

The literature has explored numerous early-stage timing optimization. For example, in global placement, strategies like net-weighting and a differentiable timing objective have been proposed for timing optimization [1], [2] However, both of them only focus on improving pre-routing timing metrics, which may have a considerable gap to sign-off timing performance. In global routing, studies have been proposed to adjust the path lengths for delay optimization [3], [4] with strategies like min-max resource sharing [5]. Besides, [6], [7] propose timing-driven layer assignment algorithms to balance routing resources and achieve better timing performance benefiting from the availability of different metal layers. Timing optimization has also been considered in the track assignment stage via nets weighting and nets detour [8], [9]. However, all the aforementioned works are not directly targeted at sign-off timing performance due to its high acquisition cost.

On the other hand, recent progress in machine learning (ML) has permitted fast and precise sign-off timing evaluation. For the first time, [10] proposes a two-stage framework with carefully selected features (e.g., pin capacitance and net length) for sign-off net delay prediction in the pre-routing stage. PERT traversals [11] are then applied to obtain the global timing metrics, i.e., endpoint slack. For further promotion, more detailed timing-relative features from a look-ahead RC network are extracted in [12]. Moreover, to directly evaluate the global timing metrics, [13] develops an end-to-end graph learning model inspired by static timing analysis (STA). Overall, these studies demonstrate the feasibility of predicting sign-off time performance with ML and open new avenues for fast and accurate early-stage timing optimization.

In this paper, we focus on explicit sign-off timing optimization at the pre-routing stage to reduce the turnaround time. As a necessary step in the most widely-used routing framework, Steiner tree construction decomposes each multi-pin net into a set of two-pin nets via additional Steiner points before global routing to reduce the problem complexity. In that case, an efficient and effective learning-assist optimization framework, TSteiner, is proposed to improve sign-off timing performance via Steiner point refinement. TSteiner makes use of graph learning to build a timing evaluator that exploits the relationship between sign-off timing metrics and Steiner point positions. On top of the learning-assist timing evaluator, an adaptive optimization framework is built to adjust Steiner point positions for better sign-off timing performance iteratively. We integrate TSteiner in the pre-routing stage of a state-of-the-art (SOTA) open-source design flow (Fig. 1) and conduct experiments on real-world designs to prove its efficacy. The major contributions of this paper are listed as follows,

- For the first time, we propose a concurrent learning-assist early-stage timing optimization framework, TSteiner, via Steiner point refinement.
- A customized graph learning framework is utilized to obtain the sign-off timing optimization gradients to guide the Steiner point refinement. Further, the proposed TSteiner framework is fully automated with an adaptive stepsize scheme and the auto-convergence scheme, which means it is not needed to manually set the stepsizes and the number of optimization iterations for designs.
- Comprehensive experiments on real-world designs show that TSteiner improves 11.2% and 7.1% on average (up to 45.8% and 43.9%) for worst negative slack and total negative slack, respectively, by integrating to the modern SOTA open-source routing flow.

The rest of this paper is organized as follows. Section II introduces timing closure, the background of Steiner points, and the problem formulation. Section III presents algorithm details of the proposed TSteiner framework. Section IV presents and analyzes the experimental results of TSteiner. Finally, Section V concludes this paper.

II. PRELIMINARIES

A. Timing Closure

Meeting the sign-off timing requirements is a critical problem in the circuit design flow to guarantee functionality correctness. Timing paths, including a startpoint and an endpoint, are extracted from the netlist for timing analysis. The startpoint can be a primary input (PI) or a register’s output pin, while the endpoint can be a primary output (PO) or a register’s input pin. Timing closure requires that the delay of each timing path must satisfy some constraints, i.e., be less than a single clock cycle. Timing violation occurs if the slack $s_e = r_e - a_e$ for a timing path endpoint $e$ is negative, where $r_e$ and $a_e$ denote $e$’s required time and arrival time, respectively. Two important metrics to
evaluate timing performance, worst negative slack (WNS) $w(\cdot)$ and total negative slack (TNS) $t(\cdot)$, can be calculated as follows,

$$w(\cdot) = \min_e s_e,$$

$$t(\cdot) = \sum_e \{\min\{0, s_e\}\}.$$  

(1)

Despite the fact that timing awareness has extended to most phases of the physical design flow, previous academic efforts have mainly focused on improving early timing metrics (e.g., evaluation from linear RC timing model and path lengths). This inspires us to find an explicit way to optimize sign-off timing performance (WNS and TNS) directly.

B. Routing and Steiner Tree

Routing is typically divided into two stages: global routing and detailed routing. In the first stage, global routing conducts rough routing on the coarse grid graph and offers guidance for the subsequent stage. Then, detailed routing works on a fine grid graph to connect all actual wires and minimize design rule violations following the guidance from global routing. Since global routing serves as a guide in the routing procedure, it may have a more significant impact on the sign-off timing performance than detailed routing.

As mentioned before, Steiner tree construction is widely-used to decompose the multi-pin net into a set of two-pin nets before global routing. The most popular Steiner minimum tree construction algorithms aim to minimize wirelength. Moreover, the Steiner point refinement is introduced to update the generated Steiner point positions for specific objectives, e.g., sign-off timing performance, while maintaining the two-pin net connections.

C. Timing Optimization via Steiner point Refinement

Generally, there are strict geometric constraints for cells, e.g., non-overlapped, since cells are real objects. Unlike cells, Steiner points work as auxiliary points in the post-placement stage; hence there are no such strict geometric constraints and pre-defined sizes for Steiner points. Furthermore, we surprisingly find that the sign-off timing performance could be significantly affected even by a random disturbance on Steiner point positions, as shown in Fig. 2.

Nevertheless, the impact of random moving is considerably unstable, and its average performance is slight (with a ratio close to 1.0). These findings demonstrate the potential of Steiner point refinement-driven timing optimization and raise the demand for a better way to guide the moving. Actually, there have been a few studies for Steiner tree-based early-stage timing optimization [3], [4] that simply consider path lengths as the objective. However, the most relevant timing metric to path lengths, net delay, does not account for much of the overall timing performance in the most widely-used technology node. Collectively, the need for more effective early-stage timing-driven Steiner point refinement algorithms is highlighted.

D. Graph Neural Networks

Graph neural networks (GNNs) have become an attractive framework for mining graph data [14]. The most popular GNNs follow an iterative message-passage scheme. Given a graph $G = (\mathcal{V}, \mathcal{E})$, a hidden embedding $h_u^{(k)}$ corresponding to each node $u \in \mathcal{V}$ is updated according to information aggregated from $u$’s graph neighborhood $\mathcal{N}(u)$ during each message-passing iteration in a GNN. The $k^{th}$ updating process can be expressed as,

$$h_u^{(k+1)} = \text{UPDATE}(k)\left(h_u^{(k)}, \text{AGGREGATE}(k)(h_v^{(k)}, \forall v \in \mathcal{N}(u))\right) = \text{UPDATE}(k)\left(h_u^{(k)}, m_{\mathcal{N}(u)}\right),$$

(2)

where $\text{UPDATE}$ and $\text{AGGREGATE}$ are differentiable functions and $m_{\mathcal{N}(u)}$ means the "message" aggregated from $u$’s graph neighborhood $\mathcal{N}(u)$. With $K$ iterations, we can define node $u$’s embedding $z_u$ as,

$$z_u = h_u^K, \forall u \in \mathcal{V}.$$  

(3)

GNNs based on the message-passing framework have shown superior efficacy in learning graph structures. Recently, GNNs have gained popularity in the electronic design automation (EDA) community [15] because the circuits can be naturally represented as graphs.

E. Problem Formulation

**Definition 1** (Timing-driven Steiner point refinement). Given an initial Steiner tree set $S_T = \{T^1, T^2, \cdots, T^n\}$, $T^i = (V^i_s, V^i_c, E^i)$, where $V^i_s$ is the set of cell nodes, $V^i_c$ is the set of Steiner nodes and $E^i$ means the edges connecting $V^i_s$ and $V^i_c$ of the $i^{th}$ Steiner tree, our task is to refine the position $(X_s, Y_s)$ of $V_s = \{V^i_s, 1 \leq i \leq n\}$ in the pre-routing stage to obtain better sign-off timing performance.

III. ALGORITHM

Before diving into the algorithm details, we first briefly introduce the overall flow of our concurrent timing-driven Steiner point refinement framework, TSteiner, as illustrated in Fig. 4. The proposed framework can be divided into two stages, sign-off timing gradient generation (Section III-A) and concurrent Steiner point refinement (Section III-B).

The flow begins by generating the initial Steiner tree set $S_T$ through modern Steiner tree construction algorithms [16], [17], which is input to a learning-assist timing evaluator $\mathcal{T}$ for point-wise gradient generation. Then a concurrent refinement process is conducted to relocate the Steiner points for better sign-off timing performance. Both two stages draw support from deep learning (DL) techniques.

Specifically, the gradient generation is inspired by DL’s forward and backward propagation, while the concurrent Steiner point refinement stage borrows ideas from the updating procedure of trainable weights.

The algorithm details are discussed in the following sections.

A. Sign-off Timing Optimization Gradient Generation

The foundation of our gradient generation framework is building the relationship between sign-off timing performance and Steiner point positions. Specifically, we utilize the trending GNN to construct an accurate sign-off timing evaluation model with Steiner point position information as input. The gradients for each Steiner point can then be generated automatically with the model’s backward propagation procedure. Specifically, the timing evaluation problem is formulated as follows,
cell edge links one input pin of a cell with its output pin, while each net edge connects a net’s drive pin to one of its sink pins. On the other hand, the Steiner graph is node-heterogeneous with two types of nodes, Steiner nodes and pin nodes, to distinguish Steiner points from pins. As for edges, it is also heterogeneous with Steiner edges and net edges.

**Definition 2. (Sign-off Timing Evaluation):** Given a Steiner tree solution $S_T$, timing evaluation is to find an estimator $T$ to evaluate the sign-off timing metrics $T(S_T)$, i.e., arrival time at each pin.

Unfortunately, all previous ML-driven pre-routing timing evaluators [10], [12], [13] did not consider Steiner points. In this paper, we design a customized model to integrate Steiner trees into the SOTA pre-routing timing prediction framework [13].

**Timing Evaluation Model Initialization** Unlike [13], which builds the graph solely from netlist connections, we construct an additional graph from the Steiner trees. The two graphs are denoted as the netlist graph and Steiner graph, respectively, as shown in Fig. 3. Specifically, the netlist graph reflects connections between pins, which is heterogeneous with two edge types: cell edge and net edge. Each cell edge links one input pin of a cell with its output pin, while each net edge connects a net’s drive pin to one of its sink pins. In the other hand, the Steiner graph is node-heterogeneous with two types of nodes, Steiner nodes and pin nodes, to distinguish Steiner points from pins. As for edges, it is also heterogeneous with Steiner edges and net edges.

**Timing Evaluation Model Inference** We propose a two-stage message-passing scheme with the above two graphs to fuse information from both Steiner trees and netlists. The scheme begins by aggregating information from the Steiner graph, which can be further divided into two steps: broadcast and reduce. In the broadcast step, information flows from each net’s drive pin to sink pins along the Steiner edges, as denoted by the purple lines in Fig. 3. During broadcasting, information on the drive pin and associated Steiner points are aggregated to each sink pin, which is then fused to update the sink pins’ features. Then in the reduce step, the updated sink pins’ features flow backward to the drive pin along the net edges to renew the drive pin’s feature, as denoted by the green lines in Fig. 3. The steps above are repeated until the Steiner tree information is fully fused. In practice, we set three iterations.

After the message-passing on the Steiner graph, the Steiner point position information, which we are interested in, has been aggregated into the related pins’ features. The updated pin features are then propagated on the netlist graph in topological order (denoted by the blue edges in Fig. 3) to generate pin node embeddings, which can be used for predicting pin-wise arrival time. Due to the page limitation, readers can refer to the propagation model in [13] for details about input features and the message-passing on the netlist graph.

Having the well-trained sign-off timing evaluator $T$, the sign-off timing metrics (WNS and TNS) can be evaluated based on the predicted endpoint arrival time, as introduced in Equation (1). Then the timing penalty can be calculated with,

$$P(T(S_T)) = \lambda_w w(T(S_T)) + \lambda_t t(T(S_T)),$$

where $w(T(S_T))$ and $t(T(S_T))$ denote the evaluated WNS and TNS. $\lambda_w$ and $\lambda_t$ are the weights for WNS and TNS, respectively.

**Timing Penalty Smoothing** As the formal formulations of WNS and TNS contain minimum or maximum operation, directly applying the
above penalty for backward propagation leads to a cut-off in some timing paths. However, timing optimization should consider all of the pins and paths globally. To overcome the above drawback, we smooth the minimum and maximum operations in the computation of WNS and TNS. To be more specific, we replace the maximum operation with the Log-Sum-Exp function LSE as follows,

\[ LSE(x_1, x_2, \cdots, x_n) = \gamma \log \left( \sum_{i=1}^{n} \exp \left( \frac{x_i}{\gamma} \right) \right), \]

where \( \gamma \) is the parameter for the degree of smoothing, and a larger \( \gamma \) indicates smoother results and lower accuracy. Similarly, the minimum operation can be treated as the maximum operation of the inverse values. Finally, the smoothed penalty function \( P_{\gamma}(\mathcal{T}(S_T)) \) can be expressed as,

\[ P_{\gamma}(\mathcal{T}(S_T)) = \lambda_w w_{\gamma}(\mathcal{T}(S_T)) + \lambda_t t_{\gamma}(\mathcal{T}(S_T)). \]

where \( w_{\gamma}(\cdot) \) and \( t_{\gamma}(\cdot) \) denote the smoothed version of \( w(\cdot) \) and \( t(\cdot) \), respectively. With the smoothed penalty, the timing optimization gradients w.r.t. Steiner points positions (\( \nabla_{X_P} P, \nabla_{Y_P} P \)) can be computed automatically via backward propagation, which is then used in our concurrent Steiner point refinement flow as described in Algorithm 1. Since we only set the feature of Steiner nodes’ positions as ‘gradient required’, no gradient will be calculated for other features, e.g., pin node positions.

B. Concurrent Timing-driven Steiner Point Refinement Framework

Having sign-off timing gradients \( \nabla_{X^{(t)}_P} P \) w.r.t. the Steiner points’ X coordinates \( X^{(t)}_i \) in \( t \)th optimization iteration, a concurrent Steiner point refinement algorithm will be applied to update \( X^{(t)}_i \) with the stochastic optimization algorithm SO described as follows,

\[ m^{(t)}_i = (1 - \beta_1) \cdot \nabla_{X^{(t-1)}_i} P_i + \beta_1 \cdot (\nabla_{X^{(t)}_i} P \otimes \nabla_{X^{(t)}_i} P) \]

\[ X^{(t)}_i = X^{(t-1)}_i - \theta \cdot \frac{m^{(t)}_i}{\sqrt{v^{(t)}_i} + \epsilon} \]

where \( \theta \) is the stepsize to optimize Steiner point positions; \( \beta_1, \beta_2, \) and \( \epsilon \) are the hyper-parameters. The update process for \( Y^{(t)}_i \) is similar to Equation (7) and shares the same hyper-parameters.

To boost the performance of our method on designs with various scales, we propose an adaptive stepsize scheme that automatically generates customized stepsize \( \theta \) fitting every design. Given the initial Steiner trees set \( S_T \), our adaptive stepsize scheme \( \text{Adaptive}_{\text{Theta}} \) can be divided into three steps:

1) Obtain the initial timing gradient (\( \nabla_{X_S} P, \nabla_{Y_S} P \)) w.r.t. the given Steiner point positions (\( X_s, Y_s \)).
2) Apply a small move:

\[ X'_s = X_s + \alpha \nabla_{X_s} P, \]
\[ Y'_s = Y_s + \alpha \nabla_{Y_s} P, \]

where \( \alpha \) is a hyper-parameter to control the scale of \( \theta \).
3) Obtain the updated timing gradient (\( \nabla_{X'_s} P, \nabla_{Y'_s} P \)).

The adaptive stepsize is then calculated as:

\[ \theta = \frac{|| (X_s - Y_s) - (X'_s - Y'_s) ||^2}{|| (\nabla_{X_S} P, \nabla_{Y_S} P) - (\nabla_{X'_S} P, \nabla_{Y'_S} P) ||^2} \]

With the adaptive stepsize scheme, our concurrent Steiner point refinement framework is described in Algorithm 1. The algorithm begins by initializing variables and setting up the optimizer (Lines 1 to 5). Following is the main part of the algorithm that conducts refinement recursively until convergence. In the \( t \)th iteration, the Steiner point positions are updated using Equation (7) to obtain the temporary Steiner trees \( S_T^{(t)} \) (Line 7). \( S_T^{(t)} \) will be stored if it achieves better (evaluated) sign-off timing performance (WNS or TNS). Otherwise, \( S_T^{(t)} \) from the previous iteration will be restored (Line 13). The optimization procedure stops when the sign-off timing metrics are fully optimized (Line 19), or it reaches the maximum optimization iterations \( \bar{N} \) (Line 16).

Algorithm 1 Concurrent Steiner point refinement. The adaptive step size scheme (\( \text{Adaptive}_{\text{Theta}} \)) and stochastic optimizer (SO) are applied to optimize the Steiner point positions using Equation (7).

Input: \( S_T \): initial Steiner trees; \( \mathcal{T} \): pre-trained timing prediction model; \( \bar{N} \): maximum optimization iterations; \( \mu \): convergence ratio.
1: \text{init}_wns ← w(\mathcal{T}(S_T)); \text{best}_wns ← w(\mathcal{T}(S_T));
2: \text{init}_tns ← t(\mathcal{T}(S_T)); \text{best}_tns ← t(\mathcal{T}(S_T));
3: \theta ← \text{Adaptive}_{\text{Theta}}(S_T);
4: \text{t} ← 0; \text{S}_T^{(0)} ← \mathcal{T}_T; \text{X}_T^{(0)} ← \text{X}_s; \text{Y}_T^{(0)} ← \text{Y}_s;
5: Initialize the optimizer SO with \( \theta \);
6: \text{repeat}
7: \text{S}_T^{(t)} ← \text{SO}(\text{S}_T^{(t-1)} \cdot (\nabla_{X^{(t)}_P} P, \nabla_{Y^{(t)}_P} P));
8: wns ← w(\mathcal{T}(\text{S}_T^{(t)})); tns ← t(\mathcal{T}(\text{S}_T^{(t)}));
9: if wns > \text{best}_wns or tns > \text{best}_tns then
10: \text{best}_wns ← wns; \text{best}_tns ← tns;
11: \text{S}_T^{(t)} ← \text{S}_T^{(t)};
12: \text{else}
13: \text{S}_T^{(t+1)} ← \text{S}_T^{(t)};
14: \text{end if}
15: \text{t} ← \text{t} + 1;
16: if \text{t} ≥ \bar{N} then
17: \text{break};
18: \text{end if}
19: until \frac{\text{init}_wns - \text{best}_wns}{\text{init}_wns} > \mu \text{ or } \frac{\text{init}_tns - \text{best}_tns}{\text{init}_tns} > \mu
20: \text{return } S_T^{(t)} \text{ (Resulting Steiner Trees)}

IV. EXPERIMENT RESULTS

A. Experiment Setting

We develop our concurrent timing optimization framework with DGL [18], PyTorch [19], and C++. The sign-off timing evaluation model is trained and tested on a Linux machine with 16 Intel Xeon Gold 6226R cores (2.90GHz), 1 GeForce RTX 3090 Ti graphics card, and 24 GB of main memory. The initial Steiner trees are generated by a widely-used Steiner minimum tree construction algorithm, FLUTE [16], followed by the edge shifting technique [17] for congestion alleviation. The subsequent routing flow is based on the modern state-of-the-art open-source routers, CUGR [20] as the global router and TritonRoute [21] as the detailed router. Both of them are running with eight threads. DrCU [22] is not used here since it cannot produce available timing analysis results with too many design rule violations on real-world designs.

We prepare ten real-world open-source designs for evaluation. All the designs are obtained from OpenCores [23], and synthesized with the widely-used open-source 130nm process design kit (PDK) [24]. In addition, we apply the industry-leading commercial tool, Cadence Innovus, to obtain the placement solution and sign-off timing reports\(^1\). The circuit benchmarks are split into training and testing sets with the benchmark statistics listed in TABLE I. The training and testing sets are determined by design scale in order to make balance. The timing evaluation model is trained on the training set with a learning rate of 5e − 4.

\(^1\)Use the Tcl command ‘timeDesign –postRoute’ with OCV mode.
TABLE I Benchmark statistics. The ten benchmarks are randomly split into the upper six benchmarks for training and the lower four for testing. ‘# Endpoints’ represents the number of timing paths.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># Nodes</th>
<th># Edges</th>
<th># Endpoints</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cell</td>
<td>Steiner</td>
<td>Net</td>
</tr>
<tr>
<td>chacha</td>
<td>15700</td>
<td>5398</td>
<td>44468</td>
</tr>
<tr>
<td>cic_decmator</td>
<td>781</td>
<td>196</td>
<td>2112</td>
</tr>
<tr>
<td>APU</td>
<td>2897</td>
<td>1154</td>
<td>8373</td>
</tr>
<tr>
<td>des</td>
<td>14652</td>
<td>5487</td>
<td>43065</td>
</tr>
<tr>
<td>jpeg_encoder</td>
<td>55264</td>
<td>15982</td>
<td>170520</td>
</tr>
<tr>
<td>spm</td>
<td>238</td>
<td>63</td>
<td>645</td>
</tr>
<tr>
<td>aes_cipher</td>
<td>11532</td>
<td>7323</td>
<td>37085</td>
</tr>
<tr>
<td>picorv32a</td>
<td>13622</td>
<td>4542</td>
<td>41030</td>
</tr>
<tr>
<td>usb_ccd_core</td>
<td>1642</td>
<td>625</td>
<td>4632</td>
</tr>
<tr>
<td>des3</td>
<td>47410</td>
<td>20004</td>
<td>136257</td>
</tr>
</tbody>
</table>

In the concurrent timing optimization flow, we initialize $\lambda_w$ as -200.0 and $\lambda_t$ as -2.0. To smooth the penalty function described in Section III-A, we set $\gamma$ as 10.0. $\alpha$ in adaptive stepsize generation is set to 5.0, and converge rate $\mu$ in the concurrent Steiner point refinement stage is set to 0.1. Starting from the 5th iteration, we increase $\lambda_w$ and $\lambda_t$ by 1% in each following iteration since the Steiner points may have already been optimized enough with 5 iterations. For each design, we constrain the largest moving distance according to the width and length of the global routing grid graph. All the reported metrics are obtained from Cadence Innovus.

B. Concurrent Timing Optimization Performance

To evaluate the timing optimization performance, we compare TSteiner against CUGR [20] and TritonRoute [21]. Experimental results compared to the routing flow without integrating TSteiner are illustrated in TABLE II. To begin with, as for the sign-off timing optimization performance, we list sign-off WNS, TNS, and the number of paths with timing violations as ‘# Vios’.

The routed wirelength ‘WL’, the number of vias ‘# Vias’, and also estimate the impact of TSteiner on the detailed routing solution quality. The routed wirelength ‘WL’, the number of vias ‘# Vias’, and also estimate the impact of TSteiner on the detailed routing solution quality. Besides the direct improvement in the sign-off timing performance, we also evaluate the improvement in the pre-routing stage and comparing it with the SOTA open-source routing flow, CUGR [20] and TritonRoute [21]. Experimental results compared to the routing flow without integrating TSteiner are illustrated in TABLE II. To begin with, as for the sign-off timing optimization performance, we list sign-off WNS, TNS, and the number of paths with timing violations as ‘# Vios’.

The routed wirelength ‘WL’, the number of vias ‘# Vias’, and the number of design rule violations ‘# DRV’ are listed. On average, our proposed framework improves the number of design rule violations by 4.51% with 0.01% more vias and 0.01% shorter routed wirelength. The significant improvement in the sign-off timing performance with a comparable routing solution quality proves the efficacy of TSteiner. Additionally, we illustrate the sign-off timing metrics ratio in Fig. 5 to prove TSteiner’s superiority compared to the random move, where we conduct the random experiments 10-50 times and then calculate the average expected value of all designs.

C. Sign-off Timing Prediction Performance

Our sign-off timing evaluation model is employed to predict the arrival time on each pin. Furthermore, we extract the predicted arrival time on the endpoints to compute the required timing metrics (WNS and TNS). The sign-off timing prediction performance on the two tasks is listed in TABLE III. Specifically, the $R^2$ score is an important metric to evaluate the coefficient of determination in statistics (the closer to 1, the better), and the formal formulation of $R^2$ score with ground truth $\{g_1, g_2, \cdots, g_n\}$ and the predicted value $\{\hat{y}_1, \hat{y}_2, \cdots, \hat{y}_n\}$ is,

$$\bar{g} = \frac{1}{n} \sum_{i=1}^{n} g_i, \quad R^2 = 1 - \frac{\sum_{i=1}^{n} (g_i - \bar{g})^2}{\sum_{i=1}^{n} (g_i - \hat{y}_i)^2}. \quad (10)$$

The results listed in TABLE III demonstrate that our proposed sign-off timing evaluation model can accurately predict sign-off timing metrics. Particularly, the $R^2$ scores of the arrival time prediction on all pins are 0.9959 in the training cases and 0.9280 in the testing cases, indicating that our timing evaluation framework well models the pin arrival time. Since the calculations of worst negative slack and total negative slack are based on the endpoints’ arrival time, we also list the $R^2$ scores for endpoint-only prediction. The results show that the proposed evaluation model consistently performs well on the endpoints.

D. Running Time

We analyze the runtime breakdown in this section since running cost is always a critical issue across the physical design flow. This paper applies the proposed TSteiner before the global routing stage. Therefore, we split the running time into three parts, TSteiner, global routing [20], and detailed routing [21]. As shown in TABLE IV, the running time of [20] is longer since we extract needed features from the Steiner tree construction stage in global routing. Furthermore, the running time of detailed routing [21] is accelerated by 6.6% since TSteiner can improve the routing quality so that less time will be spent on fixing design rule violations.

V. Conclusion

Prior works on early-stage timing optimization always simplify the problem to inaccurate early optimization due to the inability to obtain precise sign-off timing performance. TSteiner, a deep learning-assist concurrent early-stage sign-off timing optimization framework, is proposed for the first time in this study via Steiner point refinement. We confirm TSteiner’s efficacy and efficiency by integrating it into the pre-routing stage and comparing it with the SOTA open-source routing flow. The experimental results on real-world designs show that TSteiner brings 11.2% and 7.1% on average and up to 45.8% and 43.9% improvement for WNS and TNS, respectively. Overall, TSteiner can significantly improve the sign-off timing performance with only a little runtime overhead and comparable routing solution quality. Lastly, this study has raised the importance of Steiner point refinement for timing closure and provides a novel solution for early-stage timing optimization. In the future, TSteiner can be extended to more physical stages since Steiner points exist not only in the pre-routing stage but also in routing solutions.
TABLE II Experimental results on real-world open-source designs compared to the routing flow without integrating TSteiner. The sign-off timing performance and detailed routing solution quality are reported. The best one is marked with **boldface**. All the listed metrics are obtained from Innovus. The results show that TSteiner can significantly improve the sign-off timing performance with a comparable detailed routing solution quality.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>WNS (ns)</th>
<th>TNS (ns)</th>
<th># Vios</th>
<th>WL(×10^6)</th>
<th># Vias</th>
<th>DRV</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes_cipher</td>
<td>-11.246</td>
<td>-1516.9</td>
<td>512</td>
<td>984.971</td>
<td>109574</td>
<td>5</td>
</tr>
<tr>
<td>chacha</td>
<td>-48.538</td>
<td>-26259.1</td>
<td>1378</td>
<td>1,257.427</td>
<td>126600</td>
<td>2</td>
</tr>
<tr>
<td>cic_decimator</td>
<td>-2.834</td>
<td>-169.981</td>
<td>72</td>
<td>16.466</td>
<td>5586</td>
<td>3</td>
</tr>
<tr>
<td>picorv32a</td>
<td>-17.762</td>
<td>-441.607</td>
<td>67</td>
<td>727.216</td>
<td>109293</td>
<td>38</td>
</tr>
<tr>
<td>usb_cdc_core</td>
<td>-5.914</td>
<td>-1365.2</td>
<td>347</td>
<td>49.351</td>
<td>126898</td>
<td>2</td>
</tr>
<tr>
<td>APU</td>
<td>-2.265</td>
<td>-33.713</td>
<td>25</td>
<td>101.179</td>
<td>23101</td>
<td>3</td>
</tr>
<tr>
<td>des</td>
<td>-7.352</td>
<td>-405.427</td>
<td>341</td>
<td>682.828</td>
<td>115698</td>
<td>5</td>
</tr>
<tr>
<td>jpegencoder</td>
<td>-74.342</td>
<td>-64909.2</td>
<td>1967</td>
<td>2,969.654</td>
<td>439126</td>
<td>1</td>
</tr>
<tr>
<td>des3</td>
<td>-7.048</td>
<td>-1890</td>
<td>1512</td>
<td>2,680.848</td>
<td>372583</td>
<td>48</td>
</tr>
<tr>
<td>spm</td>
<td>-0.817</td>
<td>-65.866</td>
<td>126</td>
<td>4,394</td>
<td>1553</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1,000</strong></td>
<td><strong>1,000</strong></td>
<td><strong>1,000</strong></td>
<td><strong>1,000</strong></td>
<td><strong>1,000</strong></td>
<td><strong>0.888</strong></td>
</tr>
</tbody>
</table>

**TABLE III** Sign-off Timing prediction performance on two tasks, where ‘arrival-all’ and ‘arrival-ends’ represent the arrival time prediction on all pins and only endpoints, respectively.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>WNS (ns)</th>
<th>TNS (ns)</th>
<th># Vios</th>
<th>WL(×10^6)</th>
<th># Vias</th>
<th>DRV</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrival-all</td>
<td>0.9982</td>
<td>0.9980</td>
<td>0.9950</td>
<td>0.9989</td>
<td>0.9959</td>
<td>0.9991</td>
</tr>
<tr>
<td>arrival-ends</td>
<td>0.9979</td>
<td>0.9990</td>
<td>0.9977</td>
<td>0.9976</td>
<td>0.9936</td>
<td>0.9987</td>
</tr>
</tbody>
</table>

**TABLE IV** Runtime (s) breakdown

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>[20] + [21]</th>
<th>[20] + [21]</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes_cipher</td>
<td>539.847</td>
<td>16.781</td>
</tr>
<tr>
<td>chacha</td>
<td>480.123</td>
<td>19.795</td>
</tr>
<tr>
<td>cic_decimator</td>
<td>133.794</td>
<td>0.592</td>
</tr>
<tr>
<td>picorv32a</td>
<td>406.286</td>
<td>12.622</td>
</tr>
<tr>
<td>usb_cdc_core</td>
<td>54.660</td>
<td>1.247</td>
</tr>
<tr>
<td>APU</td>
<td>95.120</td>
<td>2.299</td>
</tr>
<tr>
<td>des</td>
<td>243.901</td>
<td>10.725</td>
</tr>
<tr>
<td>jpegencoder</td>
<td>893.485</td>
<td>61.433</td>
</tr>
<tr>
<td>des3</td>
<td>559.014</td>
<td>37.863</td>
</tr>
<tr>
<td>spm</td>
<td>11.848</td>
<td>0.307</td>
</tr>
<tr>
<td><strong>Ratio Avg.</strong></td>
<td><strong>1.000</strong></td>
<td><strong>1.000</strong></td>
</tr>
</tbody>
</table>

**REFERENCES**


