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[1]
[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,

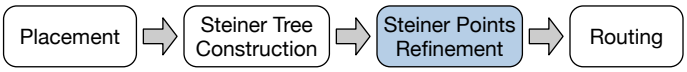

Fig. 1 Physical design flow with Steiner point refinement

- 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 stepsize 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 收敛

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

This work is supported by The Research Grants Council of Hong Kong SAR (No. CUHK14209420) and The 111 Project (No. B18001).

†Equal Contributors

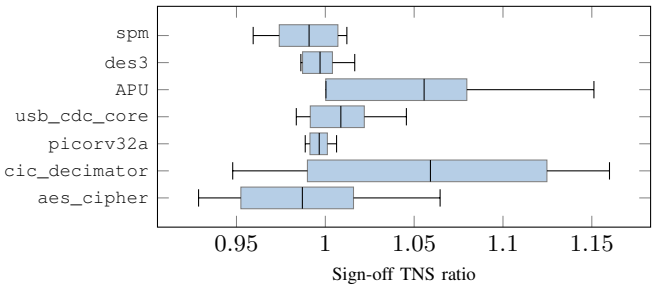

Fig. 2 Distribution of sign-off TNS ratio of the updated solution with random Steiner point disturbance to the original one (10-50 times). The larger the ratio deviates from 1, the greater the impact is.

evaluate timing performance, worst negative slack (WNS) $w(\cdot)$ and total negative slack (TNS) $t(\cdot)$, can be calculated as follows,

$$\begin{aligned} w(\cdot) &= \min_{e} s_e, \\ t(\cdot) &= \sum_{e} \{\min\{0, s_e\}\}. \end{aligned} \tag{1}$$

Despite the fact that timing awareness has been 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 considerately 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 $G = \langle \mathcal{V}, \mathcal{E} \rangle$, a hidden embedding $\boldsymbol{h}_u^{(k)}$ 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,

$$\begin{aligned} \boldsymbol{h}_u^{(k+1)} &= \text{UPDATE}^{(k)}\left(\boldsymbol{h}_u^{(k)}, \text{AGGREGATE}^{(k)}(\boldsymbol{h}_v^{(k)}, \forall v \in \mathcal{N}(u))\right) \\ &= \text{UPDATE}^{(k)}\left(\boldsymbol{h}_u^{(k)}, \boldsymbol{m}_{\mathcal{N}(u)}^{(k)}\right), \end{aligned} \tag{2}$$

where UPDATE and AGGREGATE are differentiable functions and $\boldsymbol{m}_{\mathcal{N}(u)}$ means the "message" aggregated from u's graph neighborhood $\mathcal{N}(u)$. With K iterations, we can define node u's embedding $\boldsymbol{z}_u$ as,

$$\boldsymbol{z}_u = \boldsymbol{h}_u^K, \forall u \in \mathcal{V}. \tag{3}$$

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 问题描述

Definition 1 (Timing-driven Steiner point refinement)**.** *Given an initial Steiner tree set* $S_T = \{T^1, T^2, \cdots, T^n\}$, $T^i = (V_c^i, V_s^i, E^i)$, *where* V_c^i *is the set of cell nodes,* V_s^i *is the set of Steiner nodes and* E^i *means the edges connecting* V_c^i *and* V_s^i *of the* i^{th} *Steiner tree, our task is to refine the position* (X_s, Y_s) *of* $V_s = \{V_s^i, 1 \le i \le n\}$ *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 $\mathfrak{T}$ 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.

1.初始化

2.refinement

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,

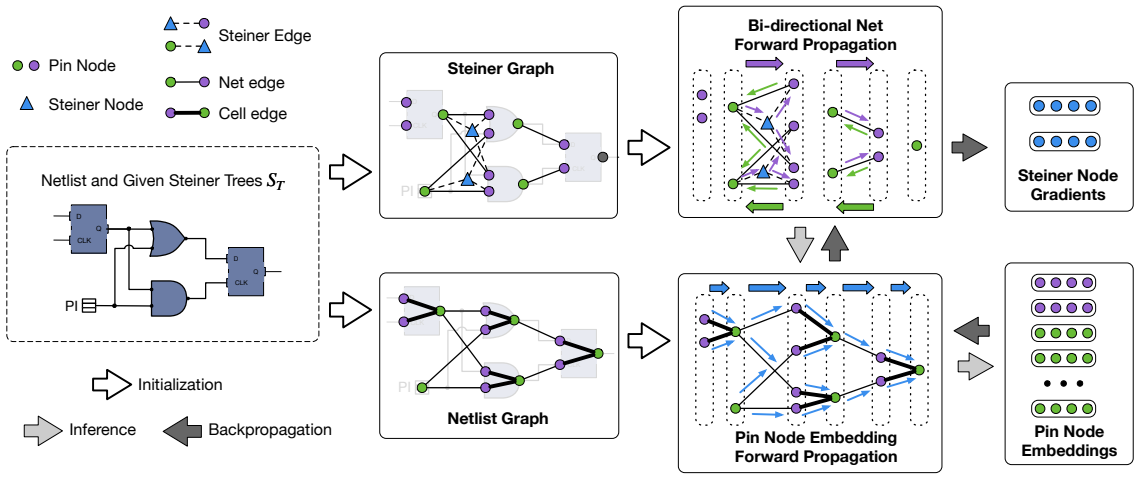

Fig. 3 Overview of our proposed gradient generation flow driven by a customized timing evaluation model. The purple and green dots represent pin nodes, while the blue triangle represents Steiner nodes. The forward inference starts from bidirectional propagation on the Steiner graph, followed by the propagation on the netlist graph. Steiner graph broadcast and Steiner graph reduce are included in the bi-directional net propagation. Further, the blue lines depict the pin embedding propagation on the netlist graph. The forward propagation is applied to obtain the pin arrival time evaluation, while the backward propagation is used to collect the Steiner node gradients.

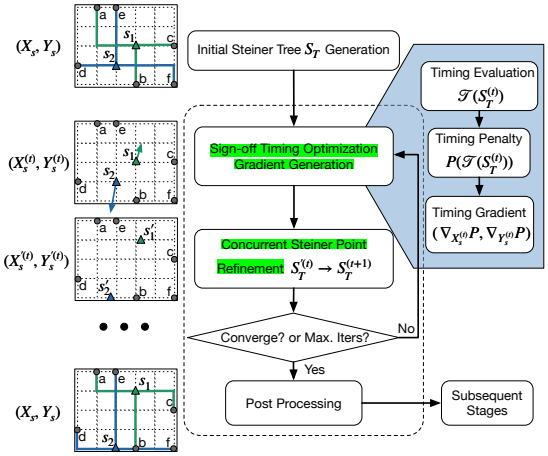

Fig. 4 The overall flow of our proposed concurrent timing optimization framework. The proposed framework includes two stages inspired by the deep model training. Note that any movement of Steiner points is constrained within the grid graph boundary, and the final positions are rounded in the post-processing. 四舍五入

Definition 2. *(Sign-off Timing Evaluation): Given a Steiner tree solution S_T, timing evaluation is to find an estimator $\mathcal{T}$ to evaluate the sign-off timing metrics $\mathcal{T}(S_T)$, 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. 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.

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 $\mathcal{T}$, 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,

$$P(\mathcal{T}(S_T)) = \lambda_w w(\mathcal{T}(S_T)) + \lambda_t t(\mathcal{T}(S_T)), \tag{4}$$

where $w(\mathcal{T}(S_T))$ and $t(\mathcal{T}(S_T))$ denote the evaluated WNS and TNS. λ_w and λ_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,

替换max，min

$$LSE(x_1, x_2, \cdots, x_n) = \gamma \log \left(\sum_{i=1}^{n} \exp \frac{x_i}{\gamma} \right), \tag{5}$$

where γ is the parameter for the degree of smoothing, and a larger γ 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,

$$P_\gamma(\mathcal{T}(S_T)) = \lambda_w w_\gamma(\mathcal{T}(S_T)) + \lambda_t t_\gamma(\mathcal{T}(S_T)). \tag{6}$$

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 $(\nabla_{X_s} P, \nabla_{Y_s} P)$ 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 $\nabla_{X_s^{(t)}} P$ w.r.t. the Steiner points' X coordinates $X_s^{(t)}$ in t^{th} optimization iteration, a concurrent Steiner point refinement algorithm will be applied to update $X_s^{(t)}$ with the stochastic optimization algorithm *SO* described as follows,

$$\begin{aligned} m_x^{(t)} &= (1-\beta_1) \cdot \nabla_{X_s^{(t)}} P, \quad v_x^{(t)} = (1-\beta_2) \cdot (\nabla_{X_s^{(t)}} P \odot \nabla_{X_s^{(t)}} P), \\ X_s'^{(t)} &= X_s^{(t)} - \theta \cdot \frac{m_x^{(t)}}{\sqrt{v_x^{(t)}} + \epsilon}, \end{aligned} \tag{7}$$

where θ is the stepsize to optimize Steiner point positions; β_1, β_2, and ϵ are the hyper-parameters. The update process for $Y_s^{(t)}$ is similar to Equation (7) and shares the same hyper-parameters.

一个亮点

To boost the performance of our method on designs with various scales, we propose an adaptive stepsize scheme that automatically generates customized stepsize θ fitting every design. Given the initial Steiner trees set S_T, our adaptive stepsize scheme *Adaptive_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:

$$\begin{aligned} X_s' &= X_s + \alpha \nabla_{X_s} P, \\ Y_s' &= Y_s + \alpha \nabla_{Y_s} P, \end{aligned} \tag{8}$$

   where α is a hyper-parameter to control the scale of θ.
3) Obtain the updated timing gradient $(\nabla_{X_s'} P, \nabla_{Y_s'} P)$.

The adaptive stepsize is then calculated as:

$$\theta = \frac{|(X_s, Y_s) - (X_s', Y_s')|_2}{|(\nabla_{X_s} P, \nabla_{Y_s} P) - (\nabla_{X_s'} P, \nabla_{Y_s'} P)|_2}. \tag{9}$$

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 N (Line 16).

解释伪代码

Algorithm 1 Concurrent Steiner point refinement. The adaptive step size scheme (*Adaptive_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; N: maximum optimization iterations; μ: converge ratio.

1: *init_wns* $\leftarrow w(\mathcal{T}(S_T))$; *best_wns* $\leftarrow w(\mathcal{T}(S_T))$;
2: *init_tns* $\leftarrow t(\mathcal{T}(S_T))$; *best_tns* $\leftarrow t(\mathcal{T}(S_T))$;
3: $\theta \leftarrow$ *Adaptive_Theta*(S_T);
4: $t \leftarrow 0$; $S_T^{(0)} \leftarrow S_T$; $X_s^{(0)} \leftarrow X_s$; $Y_s^{(0)} \leftarrow Y_s$;
5: Initialize the optimizer *SO* with θ;
6: **repeat**
7: $\quad S_T'^{(t)} \leftarrow$ *SO*$(S_T^{(t)}, (\nabla_{X_s^{(t)}} P, \nabla_{Y_s^{(t)}} P))$;
8: $\quad$ *wns* $\leftarrow w(\mathcal{T}(S_T'^{(t)}))$; *tns* $\leftarrow t(\mathcal{T}(S_T'^{(t)}))$;
9: $\quad$ **if** *wns* $>$ *best_wns* or *tns* $>$ *best_tns* **then**
10: $\quad\quad$ *best_wns* $\leftarrow$ *wns*; *best_tns* $\leftarrow$ *tns*;
11: $\quad\quad S_T^{(t+1)} \leftarrow S_T'^{(t)}$;
12: $\quad$ **else**
13: $\quad\quad S_T^{(t+1)} \leftarrow S_T^{(t)}$;
14: $\quad$ **end if**
15: $\quad t \leftarrow t + 1$;
16: $\quad$ **if** $t \geq N$ **then**
17: $\quad\quad$ break;
18: $\quad$ **end if**
19: **until** $\frac{\textit{init\_wns} - \textit{best\_wns}}{\textit{init\_wns}} > \mu$ or $\frac{\textit{init\_tns} - \textit{best\_tns}}{\textit{init\_tns}} > \mu$
20: **return** $S_T^{(t)}$ (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. Dr.CU [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[†]. 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 $5e-4$.

[†]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.

Benchmark	# Nodes		# Edges		# Endpoints
	Cell	Steiner	Net	Cell	
chacha	15700	5398	44468	41204	1972
cic_decimator	781	196	2112	1982	130
APU	2897	1154	8373	7918	427
des	14652	5487	43065	40432	2048
jpeg_encoder	55264	15982	170520	161743	4420
spm	238	63	645	516	129
aes_cipher	11532	7323	37085	35825	659
picorv32a	13622	4542	41030	38191	1879
usb_cdc_core	1642	625	4632	3999	626
des3	47410	20004	136257	125093	8872
Total Train	89532	28280	269183	253795	9126
Total Test	74206	32494	219004	203108	12036

In the concurrent timing optimization flow, we initialize λ_w as -200.0 and λ_t as -2.0. To smooth the penalty function described in Section III-A, we set γ as 10.0. α in adaptive stepsize generation is set to 5.0, and converge rate μ in the concurrent Steiner point refinement stage is set to 0.1. Starting from the 5th iteration, we increase λ_w and λ_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 with our proposed framework, TSteiner, we integrate it into the modern 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 performance, we list sign-off WNS, TNS, and the number of paths with timing violations as '# Vios'. Note that the total number of timing paths is listed in TABLE I as '# Endpoints' for each design. As shown in TABLE II, our proposed TSteiner framework can improve the sign-off WNS, TNS, and # Vios by 11.2%, 7.1%, and 3.3%, respectively. At the same time, the improvement on these three metrics can be up to 45.8%, 43.9%, and 16.4%, respectively, with only a small runtime overhead. Besides the direct improvement in the sign-off timing performance, we also estimate the impact of TSteiner on the detailed routing solution quality. 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 $\{y_1, y_2, \cdots, y_n\}$ is,

$$\bar{g} = \frac{1}{n}\sum_{i=1}^{n} g_i, \quad R^2 = 1 - \frac{\sum_i (g_i - y_i)^2}{\sum_i (g_i - \bar{g})^2}. \tag{10}$$

The results listed in TABLE III demonstrate that our proposed 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.

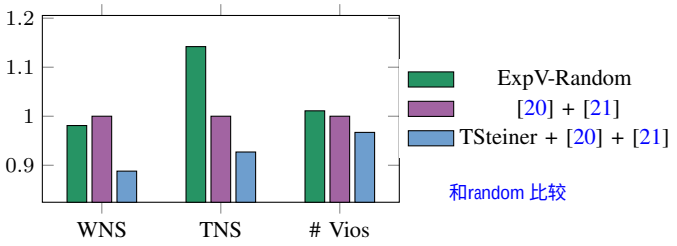

Fig. 5 Sign-off timing metrics ratio comparison. 'ExpV-Random' represents the average expected value of sign-off timing metrics with 10-50 times random moves.

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.

Benchmark	CUGR [20] + TritonRoute [21]						TSteiner + CUGR [20] + TritonRoute [21]					
	WNS (ns)	TNS (ns)	# Vios	WL($\times 10^6$)	# Vias	# DRV	WNS (ns)	TNS (ns)	# Vios	WL($\times 10^6$)	# Vias	# DRV
aes_cipher	-11.246	-1516.9	512	984.971	109574	5	**-8.38**	**-1434.2**	**504**	984.527	109443	**3**
chacha	-48.538	-26259.1	1378	1,257.427	126600	2	**-46.68**	**-25375.7**	**1372**	1,258.011	126898	2
cic_decimator	-2.834	-169.981	72	16.466	5586	3	**-2.724**	**-161.436**	72	16.413	5593	3
picorv32a	-17.762	-441.607	67	727.216	109293	38	**-17.686**	**-434.443**	**56**	727.472	109311	**37**
usb_cdc_core	-5.914	-1365.2	347	49.351	12396	0	**-5.823**	**-1343.1**	**346**	49.117	12407	0
APU	-2.265	-33.713	25	101.179	23031	3	**-2.221**	**-33.598**	25	101.454	23101	3
des	-7.352	-405.427	341	682.828	115698	5	**-3.987**	**-227.331**	**285**	682.788	115599	5
jpeg_encoder	-74.342	-64909.2	**1967**	2,969.654	439126	1	**-70.629**	**-60789.1**	2007	2,973.304	439561	1
des3	-7.048	-1890	1512	2,680.848	372583	**48**	**-5.668**	**-1879.6**	**1509**	2,684.367	372768	49
spm	-0.817	-65.866	126	4.394	1553	2	**-0.782**	**-63.846**	126	4.399	1544	2
Average	1.000	1.000	1.000	1.0000	**1.0000**	1.0000	**0.888**	**0.929**	**0.967**	**0.9999**	1.0001	**0.9549**

这么少

随便选的config？

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.

Benchmark	chacha	cic_decimator	APU	des	jpeg_encoder	spm	aes_cipher	picorv32a	usb_cdc_core	des3	Avg. Train	Avg. Test
arrival-all	0.9882	0.9980	0.9950	0.9989	0.9959	0.9991	0.9468	0.9401	0.9163	0.9087	0.9959	0.9280
arrival-ends	0.9979	0.9990	0.9977	0.9976	0.9936	0.9987	0.9459	0.9498	0.7015	0.9510	0.9974	0.8871

TABLE IV Runtime (s) breakdown

Benchmark	[20] + [21]			TSteiner + [20] + [21]			
	Total	[20]	[21]	Total	TSteiner	[20]	[21]
aes_cipher	539.847	16.781	523.066	528.512	22.222	17.830	488.460
chacha	480.123	19.795	460.328	504.902	60.331	21.642	422.929
cic_decimator	133.794	0.592	133.202	143.821	20.174	0.529	123.118
picorv32a	406.286	12.622	393.664	472.972	104.957	13.109	354.906
usb_cdc_core	54.660	1.247	53.413	115.784	59.515	1.156	55.113
APU	95.120	2.299	92.821	120.993	33.563	2.479	84.951
des	243.901	10.725	233.176	280.599	54.075	11.423	215.101
jpeg_encoder	893.485	61.433	832.052	1098.426	215.533	62.916	819.977
des3	558.014	37.863	520.151	941.474	414.908	40.841	485.725
spm	11.848	0.307	11.541	17.493	7.195	0.279	10.019
Ratio Avg.	1.000	1.000	1.000	1.320		1.017	0.934

REFERENCES

[1] P. Liao, S. Liu, Z. Chen, W. Lv, Y. Lin, and B. Yu, "DREAMPlace 4.0: timing-driven global placement with momentum-based net weighting," in *Proc. DATE*, 2022, pp. 939–944.

[2] Z. Guo and Y. Lin, "Differentiable-Timing-Driven Global Placement," in *Proc. DAC*, 2022.

[3] C. J. Alpert, A. B. Kahng, C. Sze, and Q. Wang, "Timing-driven Steiner trees are (practically) free," in *Proc. DAC*, 2006, pp. 389–392.

[4] C. J. Alpert, W.-K. Chow, K. Han, A. B. Kahng, Z. Li, D. Liu, and S. Venkatesh, "Prim-Dijkstra revisited: Achieving superior timing-driven routing trees," in *Proc. ISPD*, 2018, pp. 10–17.

[5] S. Held, D. Müller, D. Rotter, R. Scheifele, V. Traub, and J. Vygen, "Global routing with timing constraints," *IEEE TCAD*, vol. 37, no. 2, pp. 406–419, 2017.

[6] Y. Wei, Z. Li, C. Sze, S. Hu, C. J. Alpert, and S. S. Sapatnekar, "CATALYST: Planning layer directives for effective design closure," in *Proc. DATE*, 2013, pp. 1873–1878.

[7] D. Liu, B. Yu, S. Chowdhury, and D. Z. Pan, "TILA-S: Timing-driven incremental layer assignment avoiding slew violations," *IEEE TCAD*, vol. 37, no. 1, pp. 231–244, 2017.

[8] D. Wu, J. Hu, M. Zhao, and R. Mahapatra, "Timing driven track routing considering coupling capacitance," in *Proc. ASPDAC*, 2005, pp. 1156–1159.

[9] X. Gao and L. Macchiarlo, "Track routing optimizing timing and yield," in *Proc. ASPDAC*, 2011, pp. 627–632.

[10] E. C. Barboza, N. Shukla, Y. Chen, and J. Hu, "Machine learning-based pre-routing timing prediction with reduced pessimism," in *Proc. DAC*, 2019.

[11] H. Chang and S. S. Sapatnekar, "Statistical timing analysis considering spatial correlations using a single pert-like traversal," in *Proc. ICCAD*, 2003.

[12] X. He, Z. Fu, Y. Wang, C. Chang Liu, and Y. Guo, "Accurate timing prediction at placement stage with look-ahead rc network," in *Proc. DAC*, 2022.

[13] Z. Guo, M. Liu, J. Gu, S. Zhang, D. Z. Pan, and Y. Lin, "A timing engine inspired graph neural network model for pre-routing slack prediction," in *Proc. DAC*, 2022.

[14] W. L. Hamilton, "Graph representation learning," *Synthesis Lectures on Artifical Intelligence and Machine Learning*, vol. 14, no. 3, pp. 1–159, 2020.

[15] Z. Wang, C. Bai, Z. He, G. Zhang, Q. Xu, T.-Y. Ho, B. Yu, and Y. Huang, "Functionality matters in netlist representation learning," in *Proc. DAC*, 2022, pp. 61–66.

[16] C. Chu and Y.-C. Wong, "FLUTE: Fast lookup table based rectilinear steiner minimal tree algorithm for vlsi design," *IEEE TCAD*, vol. 27, no. 1, pp. 70–83, 2007.

[17] M. Pan and C. Chu, "FastRoute: A step to integrate global routing into placement," in *Proc. ICCAD*, 2006, pp. 464–471.

[18] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma *et al.*, "Deep graph library: Towards efficient and scalable deep learning on graphs," *arXiv preprint arXiv:1909.01315*, 2019.

[19] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[20] J. Liu, C.-W. Pui, F. Wang, and E. F. Young, "CUGR: Detailed-routability-driven 3d global routing with probabilistic resource model," in *Proc. DAC*, 2020, pp. 1–6.

[21] A. B. Kahng, L. Wang, and B. Xu, "Tritonroute: The open-source detailed router," *IEEE TCAD*, vol. 40, no. 3, pp. 547–559, 2020.

[22] H. Li, G. Chen, B. Jiang, J. Chen, and E. F. Young, "Dr. CU 2.0: A scalable detailed routing framework with correct-by-construction design rule satisfaction," in *Proc. ICCAD*, 2019, pp. 1–7.

[23] "OpenCores," https://opencores.org.

[24] "SkyWater Open Source PDK," https://github.com/google/skywater-pdk.