Pre-Placement Net Length and Timing Estimation by Customized Graph Neural Network

Zhiyao Xie, *Student Member, IEEE*, Rongjian Liang, Xiaoqing Xu, *Member, IEEE*, Jiang Hu, *Fellow, IEEE*, Chen-Chia Chang, Jingyu Pan, Yiran Chen, *Fellow, IEEE*

***Abstract*—Net length is a key proxy metric for optimizing timing and power across various stages of a standard digital design flow. However, the bulk of net length information is not available until cell placement, and hence it is a significant challenge to explicitly consider net length optimization in design stages prior to placement, such as logic synthesis. In addition, the absence of net length information makes accurate pre-placement timing estimation extremely difficult. Poor predictability on the timing not only affects timing optimizations but also hampers the accurate evaluation of synthesis solutions. This work addresses these challenges by a pre-placement prediction flow with estimators on both net length and timing. We propose a graph attention network method with customization, called Net^2, to estimate individual net length before cell placement[1]. Its accuracy-oriented version Net^{2a} achieves about 15% better accuracy than several previous works in identifying both long nets and long critical paths. Its fast version Net^{2f} is more than 1000× faster than placement while still outperforms previous works and other neural network techniques in terms of various accuracy metrics. Based on net size estimations, we propose the first ML-based pre-placement timing estimator. Compared with the pre-placement timing report from commercial tools, it improves the correlation coefficient in arc delays by 0.08, and reduces the mean absolute error in slack, WNS, and TNS estimations by more than 50%.**

I. Introduction

In modern VLSI design, logic synthesis plays a critical role by mapping designs in the RTL level into netlists with logic gates. Previous studies [1] show that different logic synthesis solutions can result in 3× difference in power and more than one clock cycle difference in slack when measured at the sign-off stage. As the design complexity keeps increasing, logic synthesis may not generate the netlist with the highest quality, because it lacks a credible prediction on the QoR of synthesized netlists at subsequent design stages like placement and routing (P&R). For example, estimated slacks from the synthesis tool can be largely different from sign-off static timing analysis (STA) results. To alleviate such poor predictability at the early stage, more design iterations are required to reach an optimized design quality, thus largely increase the overall turnaround time.

To improve the design predictability, a recent industrial trend among commercial EDA flows [2], [3] takes an ambitious goal to explicitly address the interaction between logic synthesis and layout. Commercial synthesis tools [4] provide increasingly better support on physical-aware logic synthesis by directly integrating both placement and optimization engines from the physical design tool [5] into its logic synthesis process. By using a unified data model in both early and late phases of the design, and by tightly integrating the engines that use that model, these state-of-the-art tools claim to achieve a tight correlation to subsequent P&R quality thus generate higher-quality netlists in a shorter turn-around time [6]. Such a trend in the EDA industry has demonstrated the importance of predictability at early design stages and its large impact on the final chip quality, but this solution is costly. Directly invoking placement and optimization engines during synthesis can be highly time-consuming. This is further discussed in detail in Section V-E.

Besides invoking core engines at downstream design stages, in recent years, machine learning (ML) techniques have been widely adopted to improve the predictability at different stages of the chip design flow. However, a large portion of these ML methods only focus on post-placement predictions. Predictions at earlier stages are more challenging due to the absence of placement information. Pre-placement ML-based works are proposed to guide synthesis flow [1], [7] or predict the power consumption [8], [9]. But existing estimators on net length, a fundamental design information related to both power and timing, still cannot achieve very high accuracy. Recent ML techniques tend to only estimate the overall wirelength of a netlist [10] or lengths of a few selected paths [11] for better accuracy, rather than predicting the length of each individual net. But the knowledge of individual net sizes can help to identify potentially large-wire nets in any path and guide transformations focusing on them. In addition, due to the absence of individual net length information before placement, the wire load cannot be accurately estimated and thus makes timing prediction extremely challenging. To the best of our knowledge, detailed pre-placement ML estimator on timing, one of the most important design objectives, is still not available until today. In summary, individual net length and timing are two important and correlated design objectives that are difficult to predict before placement. In this work, we address the problem by proposing a pre-placement prediction flow with estimators on both net length and timing.

Net Length Estimation: For state-of-the-art semiconductor manufacturing technology nodes, interconnect is a dominating factor for integrated circuit (IC) performance and power, e.g., it can contribute to over 1/3 of clock period [11] and about 1/2 of total chip dynamic power [12]. Interconnect characteristics are affected by almost every step in a design flow, but not explicitly quantified and optimized until the layout stage. Therefore, previous academic studies attempted to address the interconnect effect in design steps prior to layout, e.g., layout-aware synthesis [13], [14]. To achieve such a goal, an essential

[1]Net^2 stands for pre-placement <u>net</u> length estimator by customized graph neural <u>net</u>work

element is to enable fast yet accurate pre-layout net length prediction, which has received significant research attention in the past [10], [11], [15], [16], [17], [18], [19]. Some works [15], [16] pre-define numerous features describing each net, then a polynomial model is built by fitting these features. The work of [10] estimates wirelength by artificial neural networks (ANN), but it is limited to the total wirelength on an FPGA only, which is easier to estimate than individual net length. The mutual contraction (MC) [17] estimates net length by checking the number of cells in every neighboring net. The intrinsic shortest path length (ISPL) [18] is an interesting heuristic, which finds the shortest path between cells in the net to be estimated, apart from the net itself. The idea in [19] is similar to [18] in measuring the graph distance between cells in the netlist. The recent work [11] on wirelength prediction can only estimate the wirelength of an entire path instead of individual nets, and it relies on the results from virtual placement and routing.

Although net length prediction has been extensively studied previously, we notice a major limitation in most works. That is, they only focus on the local topology around each individual net with an over-simplified model. In other words, when estimating each net, usually their features only include information from nets one or two-hop away. The big picture, which is the net's position in the whole netlist, is largely absent. However, a placer normally optimizes a cost function defined on the whole netlist. It is not likely to achieve high accuracy without accessing any global information. Some previous models indeed attempt to embrace global information like the number of 2-pin nets in an entire circuit [15], [16], or a few shortest paths [18], but such information is either too sketchy [15], [16] or still limited to a region of several hops [18]. Since the global or long-range impact on individual nets is much more complex than local circuit topologies, it can hardly be captured by simple models or models with only human-defined parameters that cannot learn from data. To solve this, we propose a new approach, called Net$^\mathbf{2}$, based on graph attention network [20]. Its basic version, Net$^\mathbf{2f}$, intends to be fast yet effective. The other version, which emphasizes more on accuracy and is denoted as Net$^\mathbf{2a}$, captures rich global information with a highly flexible model through circuit partitioning.

Recently, deep learning has generated a huge impact on many applications where data is represented in Euclidean space. However, there is a wide range of applications where data is in the form of graphs. Machine learning on graphs is much more challenging as there is no fixed neighborhood structure like in images. All neural network-based methods on graphs are referred to as graph neural networks (GNN). The most widely-used GNN methods include graph convolution network (GCN) [21], graphSage (GSage) [22], and graph attention network (GAT) [20]. They all convolve each node's representation with its neighbors' representations, to derive an updated representation for the central node. Such operation essentially propagates node information along edges and thereby topology pattern is learned.

Similarly, in Electronic Design Automation (EDA), circuit designs are embedded in Euclidean space after placement, which inspired many convolutional neural network (CNN)-based methods [23], [24], [25]. But before placement, a circuit structure is described as a graph and spatial information is not yet available. Till recent years, GNN is explored for EDA applications [26], [27]. The work in [26] predicts observation point candidates with a model similar to GSage [22]. Graph-CNN [27] predicts the electromagnetic properties of post-placement circuits. This method is limited to very small-scale circuit graphs with less than ten nodes. Overall, GNN has great potential but is much less studied than CNN in EDA.

Timing Estimation: In digital circuit design, timing is a primary design objective that needs to be considered since very early design stages. A fast and accurate pre-placement timing estimator can essentially benefit design automation by providing early and high-fidelity feedback to synthesis solutions or during the timing-driven placement. However, accurate timing estimation is extremely challenging before placement, largely due to the absence of wire length information. It is highly difficult to estimate the impact from wires when locations of all cell instances have not been fixed. In some commercial tools [5], the timing engine ignores or under-estimates the wire load before placement. As a result, they fail to correlate well with the post-placement timing report.

ML techniques are also proposed for timing prediction. But due to aforementioned challenges, almost all existing ML-based timing estimators [28], [29], [30], [31] are only applied after placement for sign-off timing analysis. Barboza et al. [28] reduce the pessimism in the pre-routing timing report from current commercial tools. The work of [29] makes predictions with its incremental STA tools and [30] predicts sign-off timing based on non-SI (signal integrity) analysis. Besides these timing estimators, some ML-based flow tuning methods [1] optimize their flow for better timing. They typically treat the design as a black box by training one separate model for each design, and only predict the overall quality like WNS (worst negative slack) without providing any detailed timing predictions on each net or path. Compared with our method which predicts the delay at every individual net, this type of black-box predictions are significantly more coarse-grained, less challenging, and apply to fewer scenarios.

In this work, we propose to address the absence of wire information and provide an accurate pre-placement timing estimator with our knowledge from net length estimation. Both features and predictions of our net size estimator Net$^\mathbf{2}$ are selected as input to timing prediction. Different from a representative timing estimator [28] which incorporates both gate and wire delays to a net and does not differentiate multiple input pins of the same cell, we estimate the delay of every individual cell arc and net arc. To accomplish this, in our timing estimator, we construct two separate models for cell arc and net arc with different input features.

Our contributions in this work include:

- As far as we know, this is the first work making use of GNN for pre-placement net length estimation. GNNs (GCN, GSage, GAT) outperform both conventional heuristics and common machine learning methods in almost all measured metrics when validated on a comprehensive benchmark with dozens of designs.

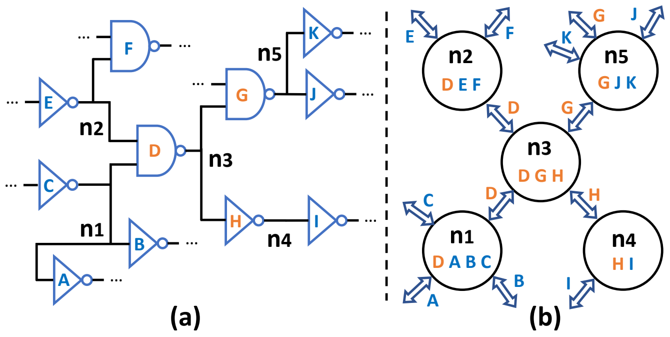


Fig. 1: (a) Part of a netlist. (b) The corresponding graph.

- We propose to extract global topology information through partitioning. Based on partition results, we define innovative directional edge features between nets, which substantially contribute to Net$^{\mathbf{2}}$'s superior accuracy.
- We propose a GAT-based model named Net$^{\mathbf{2}}$, which is customized for this net length problem. It includes a fast version Net$^{\mathbf{2f}}$, which is $1000\times$ faster than placement, and an accuracy-centric version Net$^{\mathbf{2a}}$, which effectively extracts global topology information from unseen netlists and significantly outperforms plug-in use of existing GNN techniques.
- To focus on nets, we propose a graph construction method that treats nets as nodes. In designing Net$^{\mathbf{2}}$ architecture, we define different convolution layers for graph nodes and edges to incorporate both edge and node features.
- We propose the first ML-based timing estimator before placement, to the best of our knowledge. It addresses the major challenge in pre-placement timing estimation by adopting net size estimations as input features.

II. Problem Formulation

The major target in this work is to predict the size of each net with pre-placement features. The net length L^k of each individual net n_k is the label for training and prediction. The net length is the half perimeter wirelength (HPWL) of the bounding box of the net after placement. The features of each net are based on the connection information derived from the circuit netlist. These features include information about each analyzed net's driver, sinks, fan-in size, fan-out size, and the number of neighbors. In addition, our method directly processes the netlist as a graph to capture global information of the whole circuit design.

We define terminologies of relevant features with the example in Figure 1, and commonly-used notations throughout this paper are all summarized in Table I. Figure 1(a) shows part of a netlist, including five nets $\{n_1, n_2, n_3, n_4, n_5\}$ and 11 cells $\{c_A, c_B, ..., c_K\}$. Now we focus on net n_3, which touches 3 cells $\{c_D, c_G, c_H\}$ and is referred to as a *3-pin net*. Its *driver* is cell c_D; its *sinks* are cells $\{c_G, c_H\}$. We denote the area of n_3's driver cell as a^3_{dri}. Net n_3's *fan-ins* $N^3_{in} = \{n_1, n_2\}$; its *fan-outs* $N^3_{out} = \{n_4, n_5\}$. Its *fan-in size* is 2, denoted as $|N^3_{in}| = 2$. Its *fan-out size* (number of sinks) is 2, denoted as $|N^3_{out}| = 2$. Every net can have only one driver but multiple sinks. Thus, the number of cells $= 1 + |N^3_{out}| = 3$

TABLE I: Commonly Used Notations

Notation	Description	Notation	Description
n_k	net or node	O_k	node features
$\{c_G, c_H\}$	cells	$E_{b\to k}$	edge features
a^k_{dri}	driver's area	$P[c_H]$	cell partition IDs
N^k_{in}, N^k_{out}	fan-in/fan-out nets	$M[n_k]$	node partition IDs
$h^{(t-1)}_k$, $h^{(t)}_k$	GNN embeddings	$\mathcal{N}(n_k)$	1-hop neighbors
$\lvert N^k_{in}\rvert$, $\lvert N^k_{out}\rvert$	fan-in/fan-out size	$deg(n_k)$	degree of net
$W^{(t)}$, $\theta^{(t)}$	learnable weights	$g(), \sigma()$	activation functions
$std()$, $\mu()$	std deviation, mean	L^k	net length label
$n_b \to n_k$	directional edge	[||]	concat to one list
c_{kb} or c_{bk}	cell on $n_b \to n_k$	$\backslash$	exclude from list
C2.A⇒C2.Z	timing arc	C2.A	pin A of cell C2
Net$^{\mathbf{2f/2a}}$	net size estimator	Time$^{\mathbf{f/a}}$	timing estimator

for this net. Net n_3's one-hop neighbors include both its fan-in and fan-out: $\mathcal{N}(n_3) = N^3_{in} \cup N^3_{out} = \{n_1, n_2, n_4, n_5\}$. The number of its neighbors is also known as the degree of n_3: $deg(n_3) = |\mathcal{N}(n_3)| = 4$.

To apply graph-based methods, we convert each netlist to one directed graph. Different from most GNN-based EDA tasks, net length prediction focuses on nets rather than cells. Thus we represent each net as a node, and use the terms *node* and *net* interchangeably. For each net n_k, it is connected with its fan-ins and fan-outs through their common cells by edges in both directions. The common cell shared by both nets on that edge is called its *edge cell*. For example, in Figure 1(b), net n_3 is connected with nets n_4 and n_5 through its sinks c_G and c_H; it is connected with nets n_1 and n_2 through its driver c_D. The edges through edge cell c_G is denoted as $n_3 \to n_5$ and $n_5 \to n_3$. The edge cell c_G can also be referred to as c_{35} or c_{53}. We differentiate edges in different directions because we will assign different edge features to $n_3 \to n_5$ and $n_5 \to n_3$.

An important concept throughout this paper is global and local topology information. We use the number of hops to denote the shortest graph distance between two nodes on a graph. The *information* of each net refers to its number of cells and driver's area. Local information includes the information about the estimated net itself, or from its one to two-hop neighboring nets. In contrast, global information means the pattern behind the topology of the whole netlist or the information from nets far away from the estimated net n_k. Here we define the 'far away' of global information as at least three-hop away from the analyzed net. This is beyond the scope of several previous methods [17], [16]. By performing clustering/partitioning, the global information can incorporate the information from the whole netlist, reaching the furthest net. The range of neighbors that can be accessed by each model is referred to as the model's *receptive field.*

III. Challenges

We provide an example to show the challenge in net length prediction and the importance of global information. Figure 2(a) shows a net n_6 with a commonly seen local topology

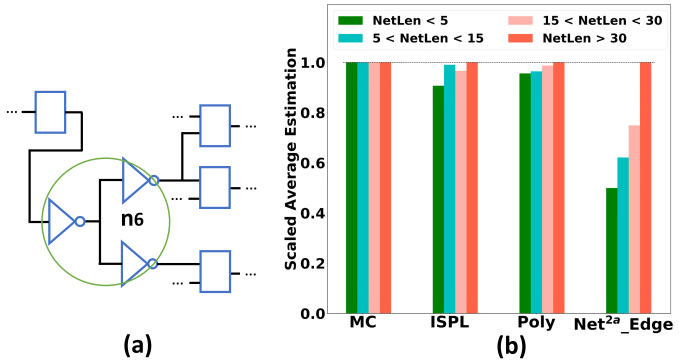


Fig. 2: (a) A typical local topology. (b) Predictions on nets with similar local topology information.

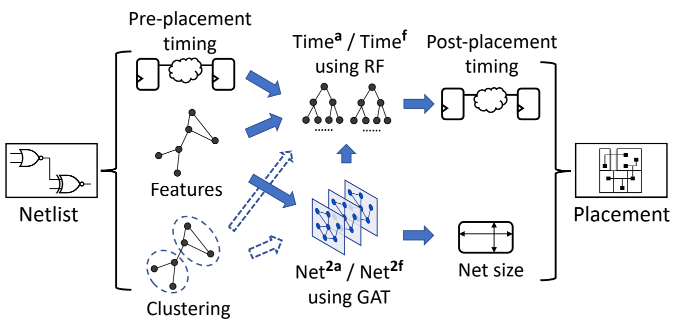


Fig. 3: The net size and timing prediction flow.

information: $|N_{in}^6| = 1$, $|N_{out}^6| = 2$, $deg(n_6) = 3$. When we inspect its neighboring nets, this net n_6 has three neighbors. These neighbors are one 2-pin net as n_6's fan-in, one 2-pin net in its fan-outs, and one 3-pin net in its fan-outs.

In a netlist of design B20 in ITC 99 [32], we find 725 nets with exactly the same driver cell's area, number of cells and one-hop neighbor information as n_6, but their net lengths after placement range from 1 µm to more than 100 µm. Distinguishing these similar nets is highly challenging without rich global information. To demonstrate this, Figure 2(b) shows the prediction from different methods on these 725 similar nets. These nets are firstly divided into four different types according to their actual net length, each type with $432, 190, 67, 36$ nets, respectively. We then plot the scaled averaged estimation by different methods for each type of net. MC [17], which only looks at one-hop neighbors, cannot distinguish these nets at all. ISPL [18], which captures some global information by searching shortest path, gives a slightly lower estimation on the shortest type (netLen $<$ 5 µm). By looking at two-hop neighbors, a polynomial model with pre-defined features (Poly) [15] [16] captures the trend with a tiny difference between different types. For Net^2, we only train its edge convolution layer on other designs and present its output. Its estimations on different net types differ significantly. This example shows the importance of global information in distinguishing a large number of nets with similar local information.

We provide a brief analysis to demonstrate why previous works like MC [17] cannot well distinguish these similar nets. According to MC [17], the mutual contraction of net $n6$ is a 3-tuple (0.4, 0.5, 0.5), contributed by the one 2-pin neighboring net and two 3-pin neighboring nets, respectively. This net length estimation by MC [17] is the same for all these 725 net with similar topology as $n6$.

IV. Algorithm

A. The Overall Flow

Figure 3 shows the overall pre-placement flow for both individual net size and timing predictions. It is applied before layout and predicts post-placement design objectives. Prediction results can benefit optimization and evaluation for both synthesis and placement. For our net length estimator Net^2, we develop a fast version and an accuracy-centric version named Net^{2f} and Net^{2a}, respectively. As Figure 3 shows, both versions of Net^2 extract features directly from the netlist, while Net^{2a} further captures global information by performing clustering on the circuit netlist. As for timing prediction, we also provide both accurate and fast versions of timing estimators, named Time^a and Time^f. The dashed blue arrows in Figure 3 mean the arrows only hold for accuracy-centric versions of our methods, like Net^{2a} and Time^a. It indicates that the clustering/partitioning information is only utilized by Net^{2a} and Time^a, providing higher accuracy at the cost of extra runtime for clustering. Besides features used by net size prediction, the pre-placement timing report from commercial EDA tools is also used as the input. The timing estimators also utilize the information from net size predictions as important input features.

Algorithm 1 Graph Generation with Node Features

Input: Basic features $\{|N_{in}^k|, |N_{out}^k|, a_{dri}^k\}$, net length label L^k, the fan-in nets N_{in}^k and fan-out nets N_{out}^k of each net n_k.
Generate Node Features:

```
1: for each net $n_k$ do
2:     $in_{in} = [\,]$, $in_{out} = [\,]$    // start with empty lists
3:     $a_{all} = [a_{dri}^k]$, $out_{in} = [\,]$, $out_{out} = [\,]$
4:     for each net $n_i \in N_{in}^k$, each net $n_o \in N_{out}^k$ do
5:         $in_{in}$.add $(|N_{in}^i|)$ ; $in_{out}$.add $(|N_{out}^i|)$
6:         $out_{in}$.add $(|N_{in}^o|)$ ; $out_{out}$.add $(|N_{out}^o|)$
7:         $a_{all}$.add$(a_{dri}^o)$
8:     $O_k = \{|N_{in}^k|, |N_{out}^k|, a_{dri}^k, \sum a_{all}, \sum out_{in},$
           $\sum out_{out}, \sum in_{in}, \sum in_{out}, std(out_{in}),$
           $std(out_{out}), std(in_{in}), std(in_{out})\}$
```

Build Graph:

```
1: Initiate a graph $G$. Each net is a node.
2: for each net $n_k$ do
3:     For node $n_k$ in $G$, set $O_k$ as node feature, $L^k$ as label.
4:     for each net $n_b \in N_{in}^k \cup N_{out}^k$ do
5:         Add directed edge $n_b \rightarrow n_k$.
```

Output: Graph G with node features O and label L.

Algorithm 2 Perform Partitioning for Edge Features

```
Input: A netlist with each net denoted as $n_k$ and each cell denoted as $c_k$. Required number of clusters/partitions.
1: Based on the netlist, construct a hyper-graph $HG_c$ with cells as nodes, nets as hyper-edges.
2: Partitioning the hyper-graph $HG_c$, result denoted as $P$. Each node(cell) $c_k$ is assigned a cluster ID $P[c_k]$.
3: Based on the netlist, construct a hyper-graph $HG_n$ with nets as nodes, cells as hyper-edges.
4: Partitioning the hyper-graph $HG_n$, result denoted as $M$. Each node(net) $n_k$ is assigned a cluster ID $M[n_k]$.
Output: Partition result $P$, $M$.
```

B. Node Features on Graph

Algorithm 1 shows how we build a directed graph and generate features for each node with a given netlist. On average, a net with more large cells tends to be longer. Thus, the most basic net features include the net's driver's area, fan-in and fan-out size $\{|N_{in}^k|, |N_{out}^k|, a_{dri}^k\}$. Feature $\sum a_{all}$ is the sum of areas over all cells in n_k. It is calculated by including the drivers of all n_k's fan-outs in line 7, which are the sinks of n_k. Besides these basic features, we capture the more complex impact from neighbors. As shown in line 4, we go through all neighbors of n_k to collect their fan-in and fan-out sizes. The summation $\sum$ and standard deviation $s()$ of these neighboring information are added to node features O_k in line 8.

C. Edge Features

In Algorithm 1, node features O_k include up to two-hop neighboring information. The receptive field of the GNN method itself depends on the model depth, which is usually two to three layers. Thus the model can reach as far as four to five-hop neighbors, which is already more than previous works. To achieve a good trade-off between accuracy, speed, and computation cost, our fast-version model Net$^{\mathbf{2f}}$ adopts this conservative and efficient setting to reach as far as five hops. But for the accuracy-centric Net$^{\mathbf{2a}}$, it goes way beyond that to capture more global information from the whole graph.

To capture global information, we use an efficient multi-level hyper-graph partitioning method hMETIS [33] to divide one netlist into multiple clusters/partitions. The partition method minimizes the overall cut between all clusters, which provides a global perspective. In this paper, we use the terms *partition* and *cluster* interchangeably. Details of this partitioning process are given in Algorithm 2. To collect more information, we construct two different types of hyper-graphs based on the netlist for this partition process. One type of hyper-graph HG_c is generated by viewing cells as nodes, and the other type of hyper-graph HG_n is generated by viewing nets as nodes. After performing partitioning with hMETIS, we get partitioning results P and M, respectively. Each net n_k is assigned a cluster ID $M[n_k]$, which denotes the cluster/partition it belongs to. Similarly, each cell c_k is assigned a cluster ID $P[c_k]$. Notice that HG_c and HG_n are only constructed to generate cluster ID for each cell and net.

Algorithm 3 Define Edge Features on Graph

```
Input: Cell cluster ID $P[c_k]$ for each cell $c_k$, net cluster ID $M[n_k]$ and the neighbors $\mathcal{N}(n_k)$ of each net $n_k$. Directed graph $G$.

1: function MEASUREDIFF($c_{bk}$, $n_b$, $c_{ok}$, $n_o$)
2:     $f_0 = 1 - (P[c_{bk}] == P[c_{ok}])$
3:     $P_b = [P[c]$ for $c \in n_b]$ // cluster IDs for $n_b$'s cells
4:     $P_o = [P[c]$ for $c \in n_o]$ // cluster IDs for $n_o$'s cells
5:     $P_{b\_not\_o} = P_b \backslash P_o$        // IDs in $P_b$ but not in $P_o$
6:     $P_{o\_not\_b} = P_o \backslash P_b$        // IDs in $P_o$ but not in $P_b$
7:     $f_1 = \frac{|P_{b\_not\_o}|}{|P_b|} + \frac{|P_{o\_not\_b}|}{|P_o|}$ // percent of different IDs
8:     $f_2 = 1 - (M[n_b] == M[n_o])$
9:     return $[f_0, f_1, f_2]$
10: end function
11:
12: for each net $n_k$ do
13:     for each net $n_b \in \mathcal{N}(n_k)$ do
14:         $F_0$ = [ ], $F_1$ = [ ], $F_2$ = [ ]
15:         Cell $c_{bk}$ is the edge cell on $n_b \rightarrow n_k$
16:         Other neighbors $N_{other}^k = \mathcal{N}(n_k) \backslash \{n_b\}$
17:         for each net $n_o \in N_{other}^k$ do
18:             Cell $c_{ok}$ is the edge cell on $n_o \rightarrow n_k$
19:             $f_0$, $f_1$, $f_2$ = MEASUREDIFF ($c_{bk}$, $n_b$, $c_{ok}$, $n_o$)
20:             $F_0$.add($f_0$) ;   $F_1$.add($f_1$) ;   $F_2$.add($f_2$)
21:         $f_3 = 1 - (M[n_b] == M[n_k])$
22:         $E_{b\rightarrow k}$ = $\{\sum F_0$, $\mu(F_0)$, $\sum F_1$, $\mu(F_1)$, $\sum F_2$,
                $\mu(F_2)$, $f_3\}$
23:         Set $E_{b\rightarrow k}$ as the feature of edge $n_b \rightarrow n_k$ in $G$.
Output: Graph $G$ with edge features $E$.
```

Cluster IDs are not directly useful by themselves. What matters in this context is the difference in cluster IDs between cells and nets. Algorithm 3 shows how the cluster information is incorporated into GNN models through novel edge features F_0, F_1, F_2, f_3. The most important intuition behind this is: for a high-quality placement solution, on average, the cells assigned to different clusters tend to be placed far away from each other.

In Algorithm 3, we design the edge features to quantify the source node's contribution to the target node's length. The contribution here means the source net is "pulling" the edge cell far away from other cells in the target net. The edge features measure such "pulling" strength. When the edge cell is "pulled" away, the target net results in a longer length. In Algorithm 3, for edge $n_b \rightarrow n_k$, function MEASUREDIFF measures the difference in assigned clusters between node n_b and every other neighboring node n_o, which indicates the distance between c_{bk} and c_{ok}. If the distance between edge cell c_{bk} and every other cell c_{ok} in n_k is large, it means c_{bk} is placed far away from other cells in net n_k. In this case, edges features F_0, F_1, F_2, f_3 are large. That is why edge features imply how strong the edge cell is "pulled" away from the target node.

Figure 4 shows an example of Algorithm 3 using the netlist

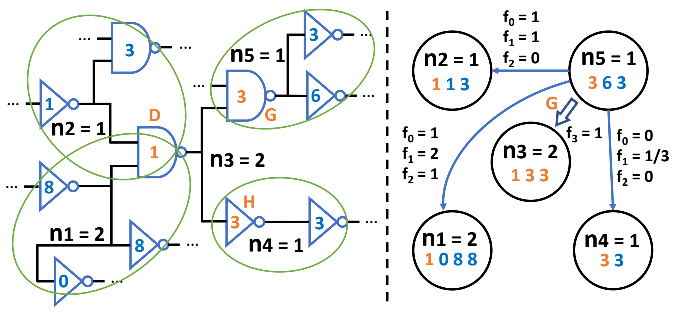


Fig. 4: Define edge features by partition results.

same as Figure 1. The number on each cell or net is the cluster ID assigned to it after partition. Figure 4 measures the edge features of edge $n_5 \rightarrow n_3$, representing how strongly edge cell c_G is pulled by n_5 from both cells $\{c_D, c_H\}$ in n_3. To calculate this, we measure the distance between c_G and c_H by MEASUREDIFF(c_G, n_5, c_H, n_4) in Algorithm 3; and the distance between c_G and c_D by MEASUREDIFF(c_G, n_5, c_D, n_1) and MEASUREDIFF(c_G, n_5, c_D, n_2).

Take MEASUREDIFF(c_G, n_5, c_H, n_4) as an example to show how it measures distance between c_G and c_H. As shown in the line 2 of Algorithm 3, feature f_0 measures the difference in c_G and c_H' cluster IDs, $f_0 = 1-(P[c_G] == P[c_H]) = 1-(3 == 3) = 0$. Feature f_1 measures the difference in all cells between n_5 and n_4. As shown from line 3 to 7, $P_5 = [3, 6, 3]$ and $P_4 = [3, 3]$. Then $P_{5\_not\_4} = [6]$ and $P_{4\_not\_5} = []$. They are normalized by the number of cells $|P_5| = 3$ and $|P_4| = 2$, in order to avoid bias toward nets with many cells. Thus, $f_1 = \frac{1}{3} + \frac{0}{2} = \frac{1}{3}$. Feature f_2 measures the difference between n_5 and n_4, $f_2 = 1 - (M[n_5] == M[n_4]) = 1 - (1 == 1) = 0$. As this example shows, we only measure whether cells / nets have the same cluster IDs, and the order of IDs does not matter.

After measuring the difference in cluster ID between c_G and all other cells in n_3, for the edge $n_5 \rightarrow n_3$, $F_0 = [1, 1, 0]$; $F_1 = [2, 1, \frac{1}{3}]$; $F_2 = [1, 0, 0]$. f_3 measures the difference between n_5 and n_3, $f_3 = 1$. This example shows how we incorporate global information from partition into edge features. Actually, we generate multiple different partitioning results M, P by requesting different number of clusters. That results in multiple different $\{F_0, F_1, F_2, f_3\}$. All these different edge features are processed in line 22 and concatenated together as the final edge features $E_{b\rightarrow k}$.

D. Common GNN Models

This section introduces how GNN models are applied on the graph G we build. GNN models are comprised of multiple sequential convolution layers. Each layer generates a new embedding for every node based on the previous embeddings. For node n_k with node features O_k, denote its embedding at the t^{th} layer as $h_k^{(t)}$. Its initial embedding is the node features $h_k^{(0)} = O_k$. Sometimes the operation includes both neighbours and the node itself, we use n_β to denote it: $n_\beta \in \mathcal{N}(n_k) \cup \{n_k\}$. In each layer t, GNNs calculate the updated embedding $h_k^{(t)}$ based on the previous embedding of the node itself $h_k^{(t-1)}$ and its neighbors $h_b^{(t-1)} | n_b \in \mathcal{N}(n_k)$.

We show one layer of GCN, GSage, and GAT below. Notice that there exist other expressions of these models. The two-dimensional learnable weight at layer t is $W^{(t)}$. In GAT, there is an extra one-dimensional weight $\theta^{(t)}$. The operation $[\;||\;]$ concatenates two vectors into one longer vector. Functions σ and g are sigmoid and Leaky ReLu activation function, respectively.

On GCN (with self-loops), $\mathcal{F}_{GCN}^{(t)}$ [21] is:

$$h_k^{(t)} = \sigma\Big(\sum_{n_\beta \in \mathcal{N}(n_k) \cup \{n_k\}} a_{k\beta} W^{(t)} h_\beta^{(t-1)}\Big)$$

$$\text{where } a_{k\beta} = \frac{1}{\sqrt{deg(k)+1}\sqrt{deg(\beta)+1}} \in \mathbb{R}$$

On GSage, $\mathcal{F}_{GSage}^{(t)}$ [22] is:

$$h_k^{(t)} = \sigma\Big(W^{(t)}\Big[h_k^{(t-1)} \Big|\Big| \frac{1}{deg(k)} \sum_{n_b \in \mathcal{N}(n_k)} h_b^{(t-1)}\Big]\Big)$$

On GAT, $\mathcal{F}_{GAT}^{(t)}$ [20] is:

$$h_k^{(t)} = \sigma\Big(\sum_{n_\beta \in \mathcal{N}(n_k) \cup \{n_k\}} a_{k\beta} W^{(t)} h_\beta^{(t-1)}\Big)$$

$$\text{where } a_{k\beta} = softmax_\beta(r_{k\beta}) \quad \text{over } n_k \text{ and its neighbors,}$$

$$r_{k\beta} = g(\theta^{(t)\intercal}[W^{(t)} h_\beta^{(t-1)} || W^{(t)} h_k^{(t-1)}]) \in \mathbb{R}$$

Here we briefly discuss the difference between these methods. GCN scales the contribution of neighbors by a pre-determined coefficient $a_{k\beta}$, depending on the node degree. GSage does not scale neighbors by any factor. In contrast, GAT uses learnable weights W, θ to firstly decide node n_β's contribution $r_{k\beta}$, then normalize the coefficient $r_{k\beta}$ across n_k and its neighbors through a softmax operation. Such a learnable $a_{k\beta}$ leads to a more flexible model. For all these GNN methods, the last layer's output embedding $h_k^{(t)}$ is connected to a multi-layer ANN.

E. Net$^{\mathbf{2}}$ Model

The node convolution layer of the Net$^{\mathbf{2}}$ is based on GAT, considering its higher flexibility in deciding neighbors' contribution $a_{k\beta}$. Thus node convolution layer is $\mathcal{F}_{GAT}^{(t)}$. In the final embedding, we concatenate the outputs from all layers, instead of only using the output of the final layer like most GNN works. This is a customization, by which the embedding includes contents from different depths. The shallower ones from the first few layers include more local information, while the deeper ones from the last few layers contain more global information. Such an embedding provides more information for the ANN model at the end and may lead to better convergence. The idea of combining shallow and deep layers has inspired many classical deep learning methods in Euclidian space [34] [35], but it is not widely applied in GNNs for node embeddings. After three layers of node convolution, the final embedding for each node is $[h_k^{(1)} || h_k^{(2)} || h_k^{(3)}]$. Without partitioning, this is the embedding for our fast solution Net$^{\mathbf{2f}}$.

In order to utilize edge features, here we define our own edge convolution layers $\mathcal{E}$ as customization. For each directed edge $n_b \rightarrow n_k$, we concatenate both target and source nodes' features $[O_k || O_b]$ together with its edge features $E_{b\rightarrow k}$ as the input of edge convolution. Combining node features when processing edge features enables $\mathcal{E}$ to distinguish different edges with similar edge features. The output embedding is:

$$e_{k\_sum} = \sum_{n_b \in \mathcal{N}(n_k)} W_2 W_1 [O_k || E_{b \to k} || O_b]$$

$$e_{k\_mean} = \frac{1}{deg(k)} e_{k\_sum}$$

The two two-dimensional learnable weights W_1 and W_2 can be viewed as applying a two-layer ANN to the concatenated input. We choose two-layer ANN rather than one-layer here because the input vector $[O_k || E_{b \to k} || O_b]$ is long and contains heterogeneous information from both edge and node. We prefer to learn from them with a slightly more complex function. After the operation, both e_{k_sum} and e_{k_mean} are on nodes. Then, we add an extra node convolution using the output from edge convolution as input. This structure learns from neighbors' edge embeddings e_{b_sum}, e_{b_mean}.

$$h_k^{(e)} = \mathcal{F}_{GAT}^{(e)}([e_{k\_sum} || e_{k\_mean}], [e_{b\_sum} || e_{b\_mean}])$$

Inspired by the same idea in Net$^{\mathbf{2f}}$, we combine the contents from all layers for our accurate solution Net$^{\mathbf{2a}}$. Its final embedding is $[h_k^{(1)} || h_k^{(2)} || h_k^{(3)} || e_{k\_sum} || e_{k\_mean} || h_k^{(e)}]$. For both Net$^{\mathbf{2f}}$ and Net$^{\mathbf{2a}}$, their final embeddings are then connected to an ANN.

F. Timing Prediction Method

This section introduces our timing prediction method in detail. The timing estimator is constructed and applied to directly predict the delay of each individual timing arc. Then based on the inference result, we further obtain arrival time, required arrival time, and slack of each circuit node by traversing the graph with predicted delay values. Similar to the Net$^{\mathbf{2}}$ model, we provide both fast and accuracy-oriented versions for timing prediction, named Time$^{\mathbf{f}}$ and Time$^{\mathbf{a}}$, respectively.

We take the simplified circuit in Figure 5 to demonstrate our timing estimator, which predicts the delay of every timing arc. The timing arc, as the basic component of a timing path, can be categorized into cell arc and net arc. Each cell arc is between an input pin and output pin of a cell, and each net arc is between the driver pin and load pin of a net. Considering their different properties, in our timing estimator, two separate timing prediction models are constructed to handle these two types of timing arcs. For each timing arc, we denote the pin from which it originates as the source pin, and the pin at which it ends as the sink pin. For example, the timing arc C2.A⇒C2.Z means a cell arc from source pin C2.A to the sink pin C2.Z.

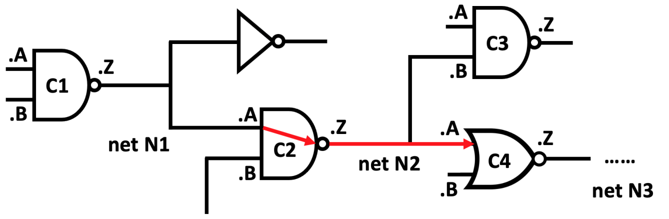


Fig. 5: An example to illustrate timing prediction algorithm.

TABLE II: Pre-Placement Features for Timing Prediction

For each cell arc (C2.A⇒C2.Z in cell C2, for example)
Pre-placement delay of the arc itself: C2.A⇒C2.Z
Source pin information: capacitance, slew, slack at C2.A
All net-size-relevant features of the following net: net N2
Predicted size of previous net: net N1

For each net arc (C2.Z⇒C4.A in net N2, for example)
Source pin information: max capacitance, slew, slack at C2.Z
Sink pin information: capacitance at C4.A
All net-size-relevant features of the net: net N2
Predicted size of the following net: net N3

For each cell, the cell-arc timing model predicts the post-placement delay of all its cell arcs. Take cell C2 in Figure 5 as an example, the model predicts delays of both C2.A⇒C2.Z and C2.B⇒C2.Z. This is essentially different from the timing model in a representative post-placement timing estimator [28], which assumes the delays of all cell arcs in the same cell are the same. This approximation in [28] may lead to inaccuracies, considering the input slews and diffusion capacitances seen by input pins of the same cell can be different. Our observation in experiments shows the cell delays at C2.A⇒C2.Z and C2.B⇒C2.Z can differ a lot and thus distinguishing all cell arcs helps to achieve higher accuracy. Similarly, for each net, the net-arc timing model predicts the post-placement delay of each net arc. In net N2, for example, there are two net arcs C2.Z⇒C4.A and C2.Z⇒C3.B. In addition, our timing estimator takes the worst delay between rising and falling as the ground truth, without constructing separate models for rising and falling edges. This avoids doubling the required timing models and simplifies the timing analysis through traversals.

Table II summarizes selected features for cell arcs and net arcs, with examples on C2.A⇒C2.Z and C2.Z⇒C4.A in Figure 5, respectively. All these features in Table II are from two main sources, as summarized below.

- All relevant slew, delay, and slack information from the pre-placement timing report.
- All relevant net and cell information that can be derived from the netlist. It includes the global information captured by performing clustering on the netlist.

Both cell-arc and net-arc models are constructed based on the existing timing report and the netlist information used in net size prediction. We can also view the prediction procedure as improving the inaccurate pre-placement timing report by incorporating net size information into our ML-based timing model. Specifically, a detailed explanation of all selected features is elaborated as follows.

- **Pre-placement delay:** Although the pre-placement timing report fails to evaluate wire load accurately for delay measurement, it still shows a generally acceptable correlation with ground truth. Thus the pre-placement delay of the predicted cell arc itself is adopted as an important input feature. Notice that the pre-placement delay of net arcs is set to zero in some commercial layout tools [5], thus the delay of the net arc itself is not used as a feature of C2.Z⇒C4.A.

- **Capacitance at the input pin of cells:** For the same type of cell, the capacitance at the cell input pin is usually proportional to the cell's driving strength. For cell arcs, a larger capacitance at each arc's source pin indicates the larger driving strength and thus smaller delay. For net arcs, a larger capacitance at each arc's sink pin indicates a higher load seen by the wire.
- **Pre-placement slew at the source pin:** The slew, or named transition time, also significantly affects the delay. Thus the pre-placement slew at the source pin of both types of arcs is adopted as features.
- **Detailed net size information:** The net size of net N2 is a determining factor of the delay. For both cell and net arcs in this example, it is directly proportional to the wire load seen by the C2.Z pin. A larger wire load at C2.Z takes longer to charge/discharge, leading to a larger cell-arc delay. For the net arc, the net size is also proportional to the wire length from C2.Z to C4.A. For the fast timing estimator $\mathrm{Time^f}$, both node features and $\mathrm{Net^{2f}}$-predicted size of the net N2 are included as features. For the accurate version $\mathrm{Time^a}$, besides using predictions from $\mathrm{Net^{2a}}$, the clustering-related information of this node is also included as features.
- **Brief net size information:** For the cell arc like C2.A⇒C2.Z, its previous net N1 affects the input slew at the source pin C2.A. For the net arc like C2.Z⇒C4.A, its following net N3 affects where cell C4 is placed, and thus affects the distance between C2.Z and C4.A. Since their impact on the arcs is less than the net N2, we only adopt the brief net size information, which is corresponding net size estimators' predictions on these nets as features.

Based on extracted features of the two different types of timing arcs, we develop one cell-arc model and one net-arc model, both based on the random forest (RF) algorithm [36]. Tree-based ML algorithms are good at handling largely distinct types of features, which include slew, delay, capacitance, cell/net number, and clustering information in this case. Instead of directly predicting the ground-truth post-placement delay of each arc, our model is actually trained to predict the *incremental delay*, which is the difference between pre-placement and the ground-truth post-placement timing. Then the final predicted delay is the summation of both pre-placement delay and the prediction of the incremental delay. This strategy, not adopted in previous ML-in-EDA works [28], helps the model to directly capture wire-load-induced delay based on the pre-placement timing report.

V. Experimental Results

A. Experimental Setup

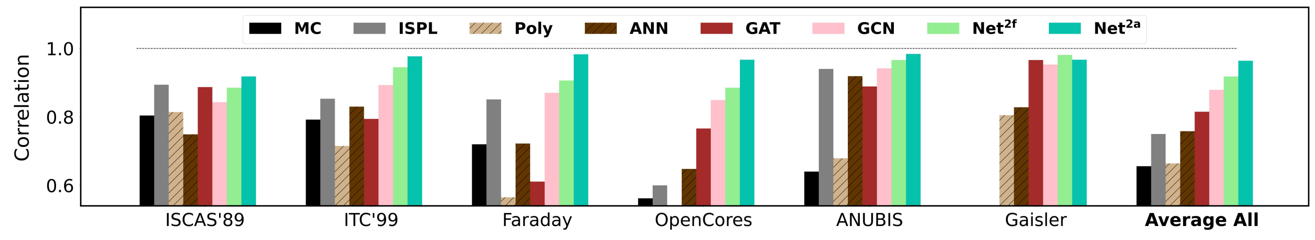
To thoroughly validate our algorithms, we constructed a comprehensive dataset by including 37 different designs with largely varying sizes. All 37 designs are synthesized with Synopsys Design Compilier® [37] in 45nm NanGate Library, and then placed by Cadence Innovus™ v17.0 [5]. When testing ML models on each design, we train the model only on the *other* 36 designs in the dataset to prevent information leakage. Thus, all accuracy numbers measure the performance on new designs completely *unseen* to the existing model. These accuracies reflect how well the model generalizes to each of the 37 designs in our dataset. The detail of each design is shown in Table III. They are collected from various benchmarks, including ISCAS'89 [38], ITC'99 [32], ANUBIS [39], and other selected designs from Faraday, OpenCores and Gaisler in the IWLS'05 [40]. To ensure all designs in the experiment are representative, we discard tiny designs with less than 3K nets. As shown in Table III, the size of these designs ranges from 4K to 800K nets. We set the clock period of all designs to be 1.5ns and thus most designs result in a negative worst slack. This mimics a common design scenario where designers target high performance and rely on the timing estimator to address negative slacks on critical paths.

TABLE III: Number of Nets in Designs

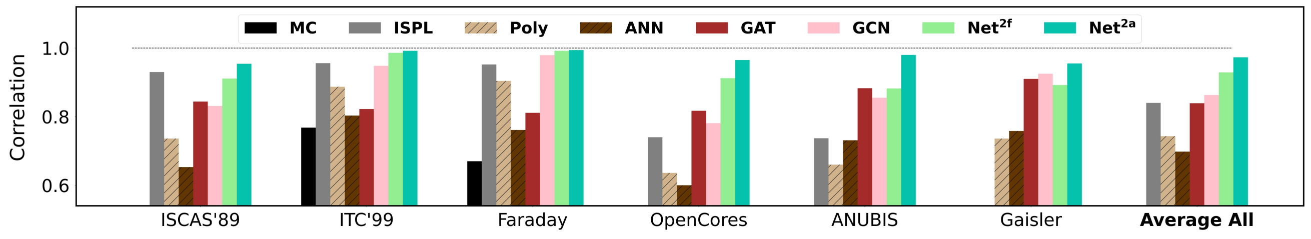
Benchmark	Design	# Net	Design	# Net
ISCAS'89	s13207	4 K	s35932	31 K
	s38417	26 K	s38584	18 K
	s5378	4 K	s9234_1	4 K
ITC'99	b14	22 K	b15	12 K
	b17	40 K	b18	115 K
	b19	225 K	b20	39 K
	b21	39 K	b22	58 K
Faraday	DMA	42 K	DSP	73 K
	RISC	98 K		
OpenCores	systemcaes	13 K	wb_dma	6 K
	systemcdes	4 K	des	6 K
	ethernet	71 K	mem_ctrl	10 K
	pci	25 K	spi	4 K
	tv80	13 K	usb_funct	24 K
	vga_lcd	106 K	wb_conmax	87 K
ANUBIS	DLX	19 K	ALPHA	41 K
	FPU	36 K	mor1k	178 K
	OR1200	847 K		
Gaisler	leon2	835 K	leon3mp	640 K
	netcard	551 K		

All GNNs are built with Pytorch 1.5 [41] and Pytorch-geometric [42]. The partition on graphs is performed by hMETIS [33] executable files. The RF models are developed based on the random forest regressor in scikit-learn [43]. The experiment is performed on a machine with a Xeon E5 processor and an Nvidia GTX 1080 graphics card.

Hyper-parameter values are decided during parameter tuning. This is accomplished by testing combinations of hyper-parameters on a much smaller validation dataset constructed for parameter tuning. This smaller validation dataset may comprise netlists only from one benchmark like ITC'99, in order to make the testing faster and allow us to test how well the model generalizes on other designs in the whole dataset. Here we introduce the best hyper-parameters after parameter tuning. They target to achieve a good trade-off between bias and variance, making the model sufficiently flexible while not too complex. For all GNN methods, we use three layers of GNN with two layers ANN. The attention head number of GAT is two. The size of each node convolution



(a)



(b)

Fig. 6: The correlation coefficient R between net length prediction and label. Averaged over designs in each benchmark. (a) 20 bins generated according to labels. (b) 20 bins generated according to predictions.

output is 64. The size of edge convolution output is twice of the input size $[O_k||E_{b\to k}||O_b]$. The size of the first-layer ANN is the same as its input embedding, and the size of the second-layer ANN is 64. A batch normalization layer is applied after each GNN layer for better convergence. Because of the difference in graph size, each batch includes only one graph, and the training data is shuffled during training. We use stochastic gradient descent (SGD) with a learning rate 0.002 and momentum factor 0.9 for optimization. GNN models converge in 250 epoches. For all RF models in timing prediction, we set the number of tree-based estimators to be 80 and the maximum depth of each estimator to be 12. Other parameters are left the same as default settings.

When partitioning each netlist, we generate seven different cell-based partitions P by requesting the number of output clusters to be the number of cells divided by 100, 200, 300, 500, 1000, 2000, and 3000. Because different partitions are generated in parallel, the overall runtime depends on the slowest one. Similarly, we generate three net-based partitions M by requesting the cluster number to be the number of nets divided by 500, 1000, and 2000. These cluster numbers are achieved by tuning during experiments, which provides good enough coverage over different cluster sizes.

Representative previous methods MC [17], ISPL [18], and Poly [16] are implemented for comparisons. As for traditional ML models, besides the polynomial model proposed in previous work [16], we implement a three-layer artificial neural network (ANN) model using node features O. Here we summarize the receptive field of all methods. MC is limited to one-hop neighbors, while Poly and ANN can reach two-hop neighbors. The receptive field of ISPL varies among different nodes. According to [18], ISPL for most nets is within several hops. In comparison, all GNNs and Net$^{\mathbf{2f}}$ can access five-hop neighbors. Net$^{\mathbf{2a}}$ measures the impact from the whole netlist.

B. Metrics for Accuracy Evaluation

We evaluate our methods with various metrics, including mean absolute error (MAE), correlation coefficient R, and coefficient of determination R^2. Given a list of ground-truth labels $\{y_i\}$ and corresponding predictions $\{p_i\}$ with length N, these metrics are defined below. The $\bar{y}$ and $\bar{p}$ represent the average of labels and predictions, respectively.

$$R = \frac{\sum_{i=1}^{N}(y_i-\bar{y})(p_i-\bar{p})}{\sqrt{\sum_{i=1}^{N}(y_i-\bar{y})}\sqrt{\sum_{i=1}^{N}(p_i-\bar{p})}}$$

$$\text{MAE} = \frac{\sum_{i=1}^{N}|y_i-p_i|}{N} \qquad R^2 = 1-\frac{\sum_{i=1}^{N}(y_i-p_i)^2}{\sum_{i=1}^{N}(y_i-\bar{y})^2}$$

In addition, for classification tasks, we evaluate the accuracy with Receiver Operating Characteristic (ROC) curve, measured based on a confusion matrix including TP (the number of true positive samples), FP (false positive), TN (true negative), and FN (false negative). Both TPR (true positive rate) and FPR (false positive rate) are defined as below.

For classification tasks, a threshold is applied to turn the raw prediction into binary. A higher threshold will lead to higher TPR, but also higher FPR; otherwise the vice. The ROC curve thus indicates the trade-off between TPR and FPR when varying the threshold, and a larger area under curve (AUC) indicates higher accuracy. The AUC ranges from 0 to 1, with AUC = 0.5 indicating the accuracy of random guessing.

C. Net Length Prediction Result

We first measure the correlation between prediction and ground truth on all nets in each netlist in Figure 6, with a classical criterion used in many net length estimation works [16], [18], [19]. For each netlist, we firstly calculate a range of net length $[L_{0\%}, L_{95\%}]$. It means from the shortest net length to the 95 percentile largest net length. The top 5% longest nets are excluded to prevent an extraordinarily large range. Then the calculated range is partitioned into 20 equal bins, and the

TABLE IV: (a) Net length prediction: long nets identification accuracy. (b) Timing prediction: arc delay prediction accuracy.

Design	(a) Long Nets Identification Accuracy in ROC AUC (%)								(b) Arc Delay Prediction Accuracy							
	MC	ISPL	Poly	ANN	GAT	GCN	$\text{Net}^{\mathbf{2f}}$	$\text{Net}^{\mathbf{2a}}$	report_timing		onlyTime		$\text{Time}^{\mathbf{f}}$		$\text{Time}^{\mathbf{a}}$	
									R	R^2	R	R^2	R	R^2	R	R^2
s13207	51.6	70.7	51.8	77.8	81.0	80.5	82.1	89.0	1.00	1.00	0.99	0.98	1.00	0.99	1.00	1.00
s35932	71.6	79.5	72.5	73.8	77.7	76.8	81.8	88.8	0.94	0.85	0.95	0.91	0.95	0.91	0.97	0.94
s38417	74.0	81.0	73.0	71.0	72.0	74.9	78.4	85.5	0.99	0.97	0.98	0.94	0.99	0.97	0.99	0.98
s38584	68.8	86.2	71.0	81.2	80.1	82.2	85.8	91.4	0.98	0.95	0.99	0.96	0.99	0.97	0.99	0.98
s5378	64.5	79.8	67.4	64.8	74.7	80.7	79.0	87.8	1.00	0.99	0.99	0.96	1.00	0.99	0.99	0.98
s9234_1	67.5	68.5	64.0	69.9	77.5	81.0	80.2	91.6	1.00	0.99	0.99	0.96	1.00	0.99	0.99	0.98
b14	70.5	77.9	71.3	74.8	74.7	76.3	81.2	90.8	0.92	0.74	0.92	0.85	0.94	0.87	0.95	0.90
b15	67.9	65.5	67.3	71.7	73.0	72.3	78.3	88.5	0.98	0.92	0.97	0.92	0.97	0.94	0.98	0.96
b17	72.4	69.0	66.3	74.0	70.2	75.5	78.7	88.8	0.96	0.87	0.95	0.89	0.97	0.93	0.97	0.95
b18*	73.2	69.1	72.4	70.6	71.9	75.4	78.9	89.6	0.96	0.88	0.96	0.91	0.96	0.90	0.98	0.95
b19*	72.0	69.2	69.4	65.3	71.8	76.0	76.5	88.6	0.96	0.86	0.92	0.83	0.92	0.73	0.97	0.93
b20	74.2	77.5	71.6	75.2	72.4	75.9	83.8	91.8	0.92	0.78	0.93	0.86	0.95	0.89	0.96	0.93
b21	74.8	76.6	73.3	77.4	75.0	78.1	84.8	92.2	0.92	0.77	0.93	0.86	0.95	0.90	0.96	0.92
b22	73.7	76.5	72.4	75.4	75.4	77.6	85.2	90.2	0.91	0.74	0.92	0.85	0.94	0.88	0.95	0.90
DMA	59.3	65.4	62.1	68.1	70.8	75.8	78.1	89.3	0.91	0.74	0.92	0.84	0.92	0.84	0.94	0.89
DSP	67.2	71.3	65.9	67.6	67.3	72.2	74.0	88.8	0.90	0.72	0.90	0.81	0.91	0.83	0.94	0.88
RISC	68.4	67.4	65.8	70.0	67.5	68.0	72.5	88.3	0.94	0.84	0.95	0.89	0.95	0.89	0.97	0.93
systemcaes	75.5	63.5	78.9	71.8	68.3	72.3	80.7	89.5	0.90	0.69	0.90	0.81	0.94	0.87	0.95	0.91
wb_dma	53.4	51.6	46.0	72.3	84.2	75.9	88.9	93.4	0.97	0.90	0.97	0.94	0.98	0.95	0.98	0.96
systemcdes	39.3	66.5	56.0	79.8	86.7	86.9	89.6	95.8	0.98	0.92	0.97	0.90	0.98	0.93	0.99	0.98
des	55.9	50.6	60.9	59.5	79.0	79.6	81.1	88.5	0.97	0.89	0.95	0.88	0.97	0.93	0.98	0.95
ethernet	65.6	67.0	65.7	66.5	78.8	76.4	80.5	85.1	0.84	0.58	0.85	0.73	0.89	0.79	0.91	0.83
mem_ctrl	57.6	61.8	57.7	64.6	77.8	75.8	80.5	88.6	0.98	0.92	0.97	0.93	0.98	0.95	0.98	0.96
pci	70.7	69.3	71.2	71.6	79.3	75.8	82.2	91.3	0.97	0.89	0.94	0.87	0.97	0.94	0.98	0.96
spi	57.0	57.4	49.6	65.6	77.3	79.7	82.7	88.1	0.98	0.94	0.98	0.95	0.99	0.97	0.99	0.97
tv80	74.2	63.9	71.6	66.3	68.9	68.6	74.4	87.3	0.96	0.87	0.96	0.91	0.97	0.93	0.97	0.94
usb_funct	69.6	68.6	70.9	72.7	77.5	73.1	79.7	92.8	0.95	0.86	0.95	0.91	0.96	0.92	0.98	0.95
vga_lcd*	67.2	55.1	77.5	79.9	86.3	88.2	88.7	93.4	0.73	0.38	0.83	0.71	0.85	0.69	0.88	0.76
wb_conmax	42.6	72.2	44.0	56.9	66.7	70.1	71.4	88.5	0.94	0.81	0.94	0.88	0.94	0.87	0.96	0.92
DLX	71.1	73.1	59.2	73.0	79.5	80.6	82.8	91.0	0.94	0.81	0.93	0.87	0.95	0.90	0.97	0.94
ALPHA	63.6	75.6	71.4	79.9	81.4	78.9	85.8	92.2	0.91	0.74	0.93	0.85	0.94	0.88	0.96	0.91
FPU	61.6	82.7	63.9	74.5	71.7	74.3	80.4	89.9	0.98	0.95	0.97	0.93	0.98	0.94	0.99	0.97
mor1k*	71.3	62.1	82.0	85.1	86.4	86.7	89.1	94.6	0.33	-0.20	0.56	0.22	0.48	0.10	0.60	0.27
OR1200*	67.9	N/A	84.1	82.7	89.0	90.1	93.8	96.1	0.53	0.10	0.80	0.48	0.75	0.35	0.81	0.51
leon2*	67.2	N/A	83.1	90.2	92.4	94.2	94.9	97.0	0.05	-0.25	0.55	0.27	0.70	0.43	0.80	0.65
leon3mp*	69.8	N/A	80.4	82.1	80.5	81.1	82.4	89.2	0.50	-0.09	0.52	-0.47	0.62	0.30	0.74	0.50
netcard*	73.7	N/A	80.6	73.1	80.5	80.2	84.0	90.1	0.35	-0.33	0.55	-0.05	0.58	0.30	0.71	0.49
Large (*) Average	**70.3**	**63.9**	**78.7**	**78.6**	**82.4**	**84.0**	**86.0**	**92.3**	**0.55**	**0.17**	**0.71**	**0.37**	**0.73**	**0.48**	**0.81**	**0.63**
Average	**66.1**	**69.1**	**67.9**	**72.9**	**76.9**	**78.0**	**82.0**	**90.4**	**0.86**	**0.70**	**0.89**	**0.78**	**0.91**	**0.82**	**0.94**	**0.87**

average of both predictions and labels in each bin is calculated. After that, the correlation coefficient R between these 20 averaged predictions and labels is measured and reported. To make fair comparisons, we calculate the range $[L_{0\%}, L_{95\%}]$ and define such 20 bins using both labels and predictions, as shown in Figure 6(a) and 6(b), respectively. Figure 6 reports the correlation coefficient averaged over designs from the same benchmark. In addition, the 'Average All' bars in Figure 6 show the averaged R over netlists from all 37 designs.

Figure 6(a) and 6(b) show highly similar trend of averaged accuracy, indicating that $\text{Net}^{\mathbf{2a}}$ > $\text{Net}^{\mathbf{2f}}$ > GCN/GAT > ANN. The correlations of GNN methods are significantly higher than previous methods with a limited receptive field including MC, ANN, ISPL, and Poly. Then $\text{Net}^{\mathbf{2f}}$ outperforms GAT and GCN with its residual connection. By capturing the global information, $\text{Net}^{\mathbf{2a}}$ performs the best on all benchmarks, with an average accuracy $R = 0.964$.

Besides correlation, we also measure the quality of net length estimators by how well they identify long nets in each circuit. Longer nets generally tend to contribute more wire load, and thus leaves a larger space for improving both wire-induced delay and the total wirelength. We believe identifying long nets is helpful for timing-related operations including timing-driven placement. Table IV(a) shows the accuracy in identifying the top 10% longest nets. For each netlist, the 10% longest nets are labeled as true, and the accuracy is measured in ROC curve's area under curve (AUC). Models capturing only one or two-hop neighbors, like MC and Poly, perform the worst. On average, ISPL outperforms MC and Poly with AUC $\approx$ 0.69. Notice that for large designs with more than 500 thousand nets, our implemented ISPL takes days of runtime, which is much slower than placement and too time-consuming in our experiment. Thus, we denote 'N/A' for these designs in Table IV(a) and omit them when measuring the average accuracy for ISPL. Using our proposed node features, the ANN achieves AUC $\approx$ 0.73. In comparison, graph methods like GCN (AUC $\approx$ 0.78) and GAT (AUC $\approx$ 0.77) perform significantly better by learning with a larger receptive field reaching five-hop neighbors. By combining shallow and deep embeddings, Net$^{\mathbf{2f}}$ achieves AUC $\approx$ 0.82. Net$^{\mathbf{2a}}$ achieves AUC $\approx$ 0.90 by learning more global information from clustering on edge features with its edge convolution layer. The trend Net$^{\mathbf{2a}}$ > Net$^{\mathbf{2f}}$ > GAT > ANN clearly decompose the contribution of different component of our ML algorithm. The good accuracy can be attributed to convolution of node features introduced in GAT, our customization of residual connections in Net$^{\mathbf{2f}}$, and the global information in Net$^{\mathbf{2a}}$.

Besides the average accuracy over all designs, we also count the average accuracy over 8 large designs with more than 100 K nets in Table IV. These large designs are marked with asterisks (*). The accuracy trend Net$^{\mathbf{2a}}$ > Net$^{\mathbf{2f}}$ > GAT > ANN remains the same for large designs. Also, the accuracy in net length prediction does not degrade when measured on large designs only.

D. *Timing Prediction Result*

For timing estimators, we first evaluate the accuracy in predicting the delay of each timing arc. Table IV(b) measures the delay of all arcs in the same netlist with both correlation coefficient R and coefficient of determination R^2. For a biased estimator, which means its predictions are consistently higher or lower than the ground-truth labels, it may achieve high R but much lower R^2 if it well correlates with the label.

In Table IV(b), the report_timing is the pre-placement timing report from the timing engine from a representative commercial tool[2]. Before placement, due to the absence of wire length information, the timing engine tends to underestimate wire load in its timing report. As a result, the reported delay values of all arcs are consistently smaller than the ground-truth post-placement report. In other words, the pre-placement report is biased towards more optimistic predictions, which is especially undesired in timing analysis since it under-estimates timing violations. Such a bias is reflected in its low averaged $R^2 = 0.70$, but the bias does not affect the correlation $R = 0.86$. When averaged over all 37 designs, the fast timing estimator Time$^{\mathbf{f}}$ is 0.05 higher in R and 0.12 higher in R^2 than the report from the commercial EDA tool. The accurate version, Time$^{\mathbf{a}}$, further achieves $R = 0.94$ and $R^2 = 0.87$. The improvement in R means both Time$^{\mathbf{f}}$ and Time$^{\mathbf{a}}$ not only

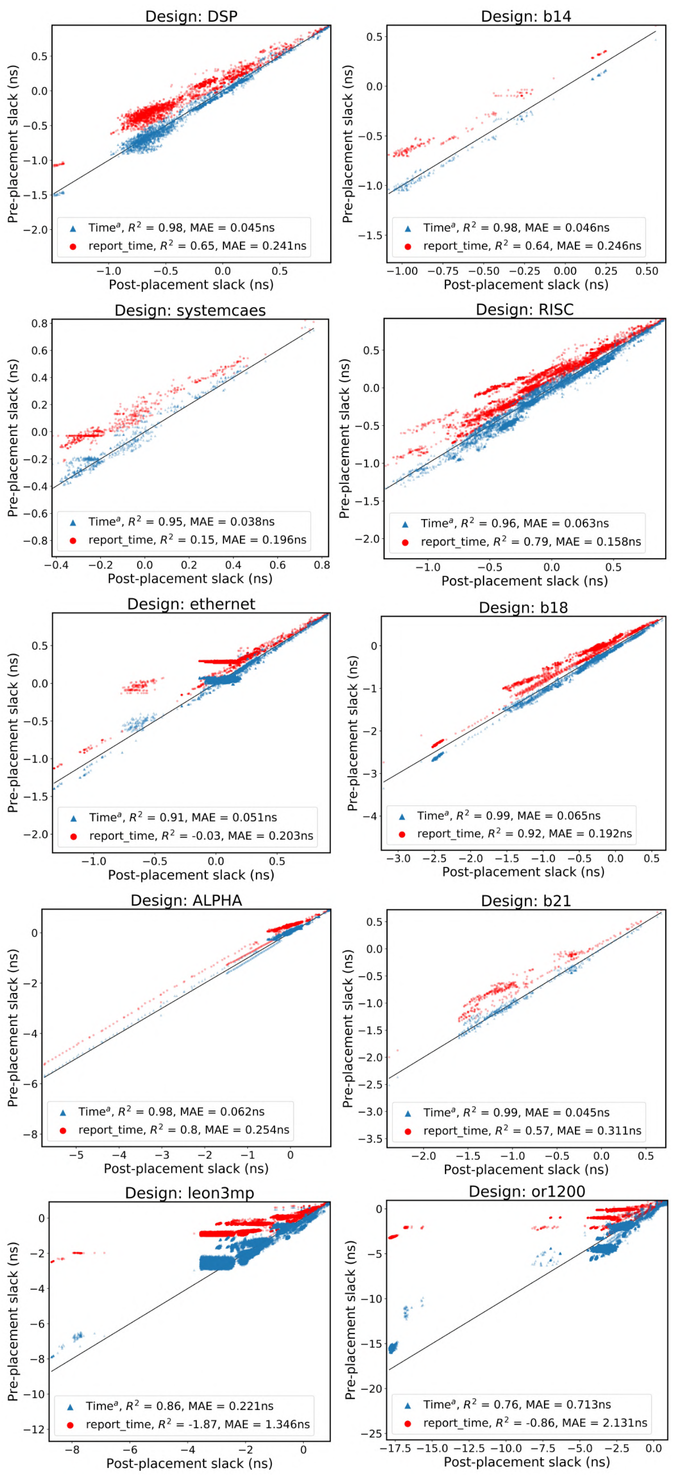


Fig. 7: Examples on pre-placement slack prediction.

fix the bias in pre-placement timing reports but also improve the correlation.

In Table IV(b), to analyze the contributions from different features, we also include an extra baseline named 'onlyTime', referring to only using timing-related features listed in Section IV-F as input features of the timing model. This baseline measures whether timing input features alone are enough for accurate timing estimations. This 'onlyTime' outperforms the

[2]According to the license agreement, we should not disclose the name of vendor's tool when making direct comparisons with it.

TABLE V: Pre-placement Path Slack Prediction Accuracy.

	report_timing		Time$^{\mathbf{f}}$		Time$^{\mathbf{a}}$	
	Error	R^2	Error	R^2	Error	R^2
Mean	**0.38 ns**	**0.39**	**0.16 ns**	**0.86**	**0.11 ns**	**0.91**
Median	**0.18 ns**	**0.77**	**0.07 ns**	**0.95**	**0.05 ns**	**0.97**

report_timing with $R = 0.89$ and $R^2 = 0.78$, but is still less accurate than Time$^{\mathbf{f}}$, and the gap is even larger compared with Time$^{\mathbf{a}}$. This gap shows the contribution purely from the net-length-related predictions. In addition, we further measured the accuracy of a timing model using the ground-truth net-length as an input feature. The averaged accuracy turns out to be $R = 0.98$ and $R^2 = 0.96$. This can be viewed as an upper limit of the current Time$^{\mathbf{f/a}}$ model, assuming perfect net length predictions are available.

We observe that the arc-delay prediction accuracy in Table IV(b) is lower for large designs with more than 100 K nets, like leon2, netcard, and OR1200. But as our analysis of Table IV(a) has demonstrated, net length prediction accuracy for large designs is not worse than average. Our study shows that in these large designs, there are much more very-long nets, which cause dominating wire-induced incremental delay. It means the gap between pre-placement and pose-placement timing is significantly larger and more difficult to predict, thus an inaccurate net length estimation causes a larger penalty to timing prediction accuracy. This is validated by the poor accuracy of 'report_timing' by the commercial tool on these large designs. Although our Time$^{\mathbf{a}}$ performs not as well on large designs, it more significantly outperforms the commercial tool baseline on large designs. As Table IV(b) shows, Time$^{\mathbf{a}}$ outperforms 'report_timing' by 0.08 (= 0.94 - 0.86) in correlation R when averaged over all designs, while by as large as 0.26 (= 0.81 - 0.55) in R for large designs.

Based on the prediction on all delay arcs, we perform the PERT [44] traversal, which is widely used in STA, to measure the arrival time, required arrival time, and slacks. Figure 7 shows predictions versus labels on calculated slacks for both pre-placement timing report and the estimator Time$^{\mathbf{a}}$. Eight representative designs are presented, and each subplot measures all slacks in the netlist. Similar to the trend of arcs delay, slacks from the pre-placement timing report are consistently higher than ground-truth, thus are biased towards optimism. For each design in Figure 7, the timing estimator Time$^{\mathbf{a}}$ achieves a much higher accuracy when measured in R^2 and absolute errors. Both averaged and median accuracies on slack prediction over all 37 designs are shown in Table V. The averaged accuracy is more affected by less accurate predictions thus the median accuracy is higher. The trend in accuracy remains the same, showing Time$^{\mathbf{a}}$ > Time$^{\mathbf{f}}$ > report_timing. On average, the Time$^{\mathbf{a}}$ achieves high $R^2 = 0.91$, indicating that high correlation and low bias are achieved simultaneously. Its mean absolute error is 0.11ns, which reduces the error in pre-placement timing report by more than 50% and is less than 10% of the clock cycle.

According to all slacks calculated by traversing delay predictions, we can easily measure the total negative slack (TNS) and worst negative slack (WNS) of each netlist. The TNS and WNS of all designs are presented in Figure 8. For the small portion of netlists with all slacks positive, we set the WNS to the lowest positive slack and leave the TNS to be zero. Each point in Figure 8 represents the TNS/WNS of one netlist. The estimator Time$^{\mathbf{a}}$ maintains its high accuracy in TNS and WNS prediction. Considering this correlation is measured on all designs and each testing design is completely unseen by the model, the result proves that the performance of Time$^{\mathbf{a}}$ is robust on all 37 tested designs in our experiment.

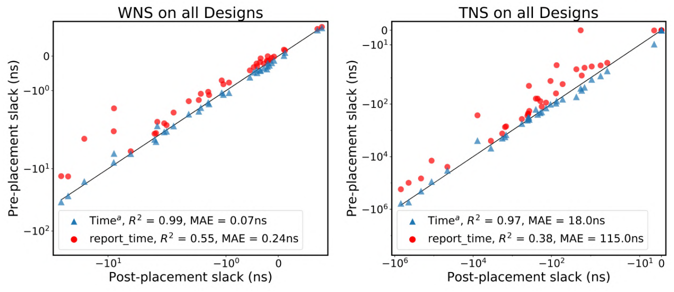


Fig. 8: WNS (left) and TNS (right) of all designs.

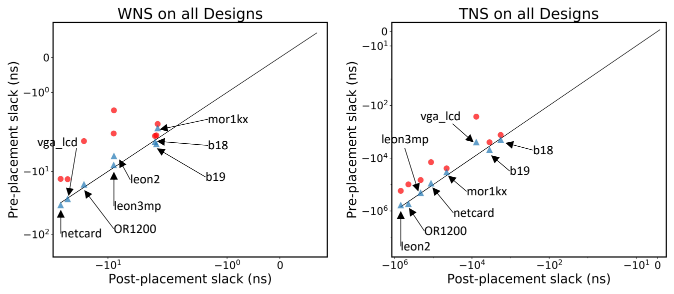


Fig. 9: WNS (left) and TNS (right) on only large designs.

To show our timing model's performance on large designs more clearly, we pick those 8 largest designs with more than 100K nets and only show the TNS/WNS predictions on these designs in Figure 9. The arrow with design name text in Figure 9 points to the prediction from Time$^{\mathbf{a}}$, and the corresponding evaluation from report_timing shares the same ground-truth in the x-axis. Compared with other designs, WNS/TNS of large designs is more negative. The estimation from report_timing is close to ground-truth for designs 'b18' and 'b19', but significantly more optimistic for designs 'vga_lcd', 'netcard', 'OR1200', and 'leon2'. In comparison, our Time$^{\mathbf{a}}$ gives rather accurate predictions to all these large designs.

E. Runtime Comparison

Table VI shows the runtime of placement, net length estimators Net$^{\mathbf{2f/2a}}$, and timing estimators Time$^{\mathbf{f/a}}$. We report the runtime separately for multiple representative designs, which cover a large range of design sizes from 12K to 800K nets in the netlist. For a fair comparison, the runtime of placement includes the placement algorithm only, without any extra time for file I/O, floorplanning, or placement optimization. The

TABLE VI: Detailed Runtime Comparison on Representative Designs (In Seconds)

Design	# Net	Placement	Partition	$\text{Net}^{\mathbf{2f}}$ Infer	$\text{Net}^{\mathbf{2a}}$ Infer	$\text{Net}^{\mathbf{2f}}$ Speedup	$\text{Net}^{\mathbf{2a}}$ Speedup	$\text{Time}^{\mathbf{f}}$ Infer	$\text{Time}^{\mathbf{a}}$ Infer	$\text{Time}^{\mathbf{f}}$ Speedup	$\text{Time}^{\mathbf{a}}$ Speedup	Extraction Overhead
b15	12 K	30	1.6	0.03	0.03	1K×	18×	0.08	0.08	0.3K×	18×	3.5
b21	39 K	128	7.1	0.05	0.05	2.6K×	18×	0.25	0.27	0.4K×	17×	8.6
mor1k	178 K	1174	64.8	0.11	0.25	11K×	18×	0.97	1.1	1.1K×	18×	43
netcard	551 K	5517	313	0.48	1.0	11K×	18×	2.9	5.8	1.6K×	17×	134
leon3mp	640 K	7180	283	0.54	1.0	13K×	25×	3.8	5.9	1.7K×	25×	156
OR1200	847 K	11353	427	0.67	1.5	17K×	26×	5.8	9.4	1.8K×	26×	214
leon2	835 K	11544	428	0.67	1.4	17K×	27×	5.0	9.8	2.0K×	26×	208

TABLE VII: Synthesis Runtime Measurement (In Seconds)

Traditional Synthesis	Placement	Partition
167	128	9.4
Physical-aware Synthesis (with Fast Placement)	Physical-aware Synthesis (with Complete Placement)	
282	414	

inference of $\text{Net}^{\mathbf{2f}/\mathbf{2a}}$ requires one Nvidia GTX 1080 graphics card, and other runtimes are performed with CPU only.

As Table VI shows, $\text{Net}^{\mathbf{2a}}$ takes slightly longer inference time than $\text{Net}^{\mathbf{2f}}$ for its extra edge convolution layer. The overall runtime of $\text{Net}^{\mathbf{2a}}$ includes both partition and inference. Partitioning contributes the majority of $\text{Net}^{\mathbf{2a}}$'s runtime. $\text{Net}^{\mathbf{2a}}$ is more than $> 15\times$ faster than placement. The runtime of $\text{Net}^{\mathbf{2a}}$ can be potentially improved by using coarser but faster partition P and M, especially on larger designs. Without partition, $\text{Net}^{\mathbf{2f}}$ is $> 1000\times$ faster than placement. For timing estimators, similarly, $\text{Time}^{\mathbf{a}}$ takes longer inference time than $\text{Time}^{\mathbf{f}}$ since it takes more input features. The $\text{Time}^{\mathbf{a}}$ is $> 15\times$ faster and the $\text{Time}^{\mathbf{f}}$ is $> 1000\times$ faster than the placement for not-too-small designs. In addition, we report the overhead time to extract all features from the circuit raw data in Table VI. The overhead is now comparable with the cost of partition. Currently this feature extraction step is implemented only to verify our ML algorithm and not optimized for fast runtime yet. It dumps all feature information from the EDA tools and then loads it to the external ML model. The room for improvement is large if extra engineering effort is spent to integrate the flow into industrial tools. The runtime comparison between different designs in Figure VI shows that the speedup of $\text{Net}^{\mathbf{2f}/\mathbf{2a}}$ and $\text{Time}^{\mathbf{f}/\mathbf{a}}$ is more significant for larger designs. It validates the scalability of our method.

Besides comparisons with placement, to gain more insights on the whole VLSI design flow, we also evaluate the runtime of both traditional logic synthesis and physical-aware synthesis from an industry-standard commercial synthesis tool[3] in Table VII. It is measured on the design b21. As introduced, physical-aware synthesis explicitly addresses the interaction between synthesis and layout with the cost of extra runtime. The commercial synthesis tool we use offers two options for physical-aware synthesis, one using a fast placement method and the other using the complete placement to provide feedback on the backend implementations. As shown in Table VII, the two versions of physical-aware logic synthesis take 115 and 247 more seconds than traditional logic synthesis. This extra runtime is close to the time spent on placement. This verifies our claim that compared with ML-based solutions, the physical-aware synthesis is more time-consuming.

[3]According to the license agreement, we should not disclose the name of vendor's tool when making direct comparisons with it.

As for the model training time, it takes around 30 minutes to train the $\text{Net}^{\mathbf{2a}}$ model, and less than 10 minutes to train the RF-based timing estimator $\text{Time}^{\mathbf{a}}$.

VI. Conclusion

In this paper, we propose $\text{Net}^{\mathbf{2}}$, a graph attention network method customized for individual net length estimation. It includes a fast version $\text{Net}^{\mathbf{2f}}$ which is $1000\times$ faster than placement, and an accuracy-centric version $\text{Net}^{\mathbf{2a}}$ which extracts global information and significantly outperform all previous net length estimation methods. Based on net length predictions, we further develop a pre-placement timing estimator, which achieves significantly better correlations with ground truth compared with the pre-placement timing report from commercial tools.

References

[1] Z. Xie, G.-Q. Fang, Y.-H. Huang, H. Ren, Y. Zhang, B. Khailany, S.-Y. Fang, J. Hu, Y. Chen, and E. C. Barboza, "Fist: A feature-importance sampling and tree-based method for automatic design flow parameter tuning," in *ASPDAC*, 2020.

[2] Synopsys, "Fusion compiler: the singular rtl-to-gdsii digital implementation solution," 2020, Accessed on: May 2021. [Online]. Available: https://www.synopsys.com/implementation-and-signoff/physical-implementation/fusion-compiler.html

[3] Cadence, "Cadence digital full flow optimized to deliver improved quality of results with up to 3x faster throughput," 2020, Accessed on: May 2021. [Online]. Available: https://www.cadence.com/en_US/home/company/newsroom/press-releases/pr/2020/cadence-digital-full-flow-optimized-to-deliver-improved-quality-.html

[4] ——, "Cadence genus user guide," 2019, Accessed on: May 2021. [Online]. Available: https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/digital-design-signoff/genus-synthesis-solution-ds.pdf

[5] ——, "Cadence innovus user guide," 2017, Accessed on: May 2021. [Online]. Available: https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/digital-design-signoff/innovus-implementation-system-ds.pdf

[6] S. Kapoor and M. Richards, "Getting better results faster with the singular rtl-to-gdsii product," 2021, Accessed on: May 2021. [Online]. Available: https://www.synopsys.com/implementation-and-signoff/resources/articles/unified-data-engines-fusion-compiler.html

[7] C. Yu, H. Xiao, and G. De Micheli, "Developing synthesis flows without human knowledge," in *ICCAD*, 2018.

[8] Y. Zhou, H. Ren, Y. Zhang, B. Keller, B. Khailany, and Z. Zhang, "PRIMAL: Power Inference using Machine Learning," in *DAC*, 2019.

[9] Y. Zhang, H. Ren, and B. Khailany, "GRANNITE: Graph Neural Network Inference for Transferable Power Estimation," in *DAC*, 2020.

[10] Q. Liu, J. Ma, and Q. Zhang, "Neural network based pre-placement wirelength estimation," in *FPT*, 2012.
[11] D. Hyun, Y. Fan, and Y. Shin, "Accurate wirelength prediction for placement-aware synthesis through machine learning," in *DATE*, 2019.
[12] N. Magen, A. Kolodny, U. Weiser, and N. Shamir, "Interconnect-power dissipation in a microprocessor," in *SLIP*, 2004.
[13] M. Pedram and N. Bhat, "Layout driven technology mapping," in *DAC*, 1991.
[14] M. Pedram and N. B. Bhat, "Layout driven logic restructuring/decomposition," in *ICCAD*, 1991.
[15] S. Bodapati and F. N. Najm, "Prelayout estimation of individual wire lengths," *TVLSI*, 2001.
[16] B. Fathi, L. Behjat, and L. M. Rakai, "A pre-placement net length estimation technique for mixed-size circuits," in *SLIP*, 2009.
[17] B. Hu and M. Marek-Sadowska, "Wire length prediction based clustering and its application in placement," in *DAC*, 2003.
[18] A. B. Kahng and S. Reda, "Intrinsic shortest path length: a new, accurate a priori wirelength estimator," in *ICCAD*, 2005.
[19] Q. Liu and M. Marek-Sadowska, "Pre-layout wire length and congestion estimation," in *DAC*, 2004.
[20] P. Veličković *et al.*, "Graph attention networks," in *ICLR*, 2017.
[21] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *ICLR*, 2016.
[22] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *NeurIPS*, 2017.
[23] Z. Xie *et al.*, "Routenet: Routability prediction for mixed-size designs using convolutional neural network," in *ICCAD*, 2018.
[24] Y.-C. Fang *et al.*, "Machine-learning-based dynamic ir drop prediction for eco," in *ICCAD*, 2018.
[25] Z. Xie, H. Ren, B. Khailany, Y. Sheng, S. Santosh, J. Hu, and Y. Chen, "Powernet: Transferable dynamic ir drop estimation via maximum convolutional neural network," in *ASPDAC*, 2020.
[26] Y. Ma *et al.*, "High performance graph convolutional networks with applications in testability analysis," in *DAC*, 2019.
[27] G. Zhang, H. He, and D. Katabi, "Circuit-gnn: Graph neural networks for distributed circuit design," in *ICML*, 2019.
[28] E. C. Barboza, N. Shukla, Y. Chen, and J. Hu, "Machine learning-based pre-routing timing prediction with reduced pessimism," in *DAC*, 2019.
[29] A. B. Kahng, S. Kang, H. Lee, S. Nath, and J. Wadhwani, "Learning-based approximation of interconnect delay and slew in signoff timing tools," in *SLIP*, 2013.
[30] A. B. Kahng, M. Luo, and S. Nath, "Si for free: machine learning of interconnect coupling delay and transition effects," in *SLIP*, 2015.
[31] S.-S. Han, A. B. Kahng, S. Nath, and A. S. Vydyanathan, "A deep learning methodology to proliferate golden signoff timing," in *DATE*, 2014.
[32] F. Corno, M. S. Reorda, and G. Squillero, "Rt-level itc'99 benchmarks and first atpg results," *Design & Test of computers*, 2000.
[33] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: applications in vlsi domain," *TVLSI*, 1999.
[34] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016.
[35] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *MICCAI*, 2015.
[36] L. Breiman, "Random forests," *Machine learning*, 2001.
[37] Synopsys, "Synopsys design compilier user guide," 2018, Accessed on: May 2021. [Online]. Available: https://www.synopsys.com
[38] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *ISCAS*, 1989.
[39] R. T. Possignolo, N. Kabylkas, and J. Renau, "Anubis: A new benchmark for incremental synthesis," in *Int. Workshop Logic Synthesis*, 2017.
[40] C. Albrecht, "Iwls 2005 benchmarks," in *IWLS*, 2005.
[41] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *NeurIPS*, 2019.
[42] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR-W*, 2019.
[43] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *JMLR*, 2011.
[44] H. Chang and S. S. Sapatnekar, "Statistical timing analysis considering spatial correlations using a single pert-like traversal," in *ICCAD*, 2003.