Disentangle, Align and Generalize: Learning A Timing Predictor from Different Technology Nodes[*]

Xinyun Zhang[†]
CUHK

Binwu Zhu[†]
CUHK

Fangzhou Liu
CUHK

Ziyi Wang
CUHK

Peng Xu
CUHK

Hong Xu
CUHK

Bei Yu
CUHK

Abstract

In VLSI design, accurate pre-routing timing prediction is paramount. Traditional machine learning-based methods require extensive data, posing challenges for advanced technology nodes due to the time-consuming data preparation. To mitigate this issue, we propose a novel transfer learning framework that uses data from previous nodes for learning on the target node. Our method initially disentangles and aligns timing path features across different nodes, then predicts each path's arrival time employing a Bayesian-based model capable of handling highly variable arrival time and generalizing to new designs. Experimental results on transfer learning from 130nm to 7nm nodes validate our method's effectiveness.

1 Introduction

Modern chips impose stringent demands on timing constraints. To comply with these rigorous timing constraints, placement and routing (PnR) processes are frequently performed in an iterative manner, which can be highly time-consuming. As a result, researchers are endeavoring to predict the timing report before the routing step. This 'look-ahead' mechanism provides preliminary feedback for timing optimization, potentially expediting the chip design process.

The widely-used timing prediction approach is the linear RC static timing analysis (STA) model, e.g., Elmore's model [1], which quickly evaluates timing using placement results. However, its efficacy is compromised due to the absence of routing information. Recent studies have achieved remarkable achievements for pre-routing timing prediction by leveraging machine learning (ML) methodologies. Barboza *et al.* [2] predict local net/cell delay and slew from high-level placement result features. Guo *et al.* [3] propose an end-to-end graph neural network (GNN) for predicting pre-routing arrival time and slack values at timing endpoints. Considering the impact of timing optimization, Wang *et al.* [4] develop an endpoint embedding framework that integrates both netlist and layout information.

[*]This work is partially supported by The Research Grants Council of Hong Kong SAR (No. CUHK14210723).
[†]Both authors contributed equally to this research.

DAC '24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3656251

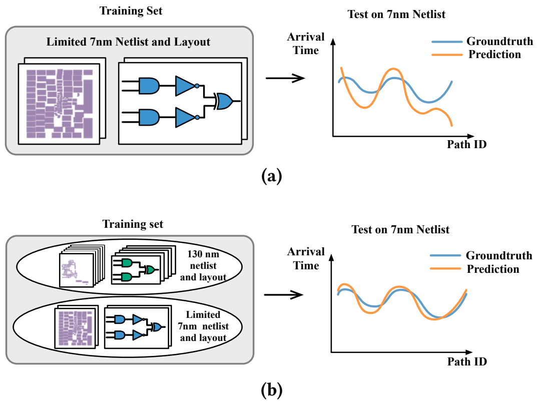

Figure 1: (a) Trained on limited 7nm netlist data; (b) Trained on both limited 7nm netlist data and 130nm netlist data.

Albeit great success has been made, the training of a precise and highly generalizable ML-based timing model necessitates a substantial volume of training data at the target technology node. When only trained on limited data, the timing predictor may suffer from severe performance drop, as shown in Figure 1(a). However, with the rapid development of the technology node, collecting complete timing data requires running a series of tools, which may be extremely time-consuming for extensive data. This presents substantial challenges for training a timing predictor for the advanced technology node. To address this issue, we propose a transfer learning framework that leverages extensive data at the preceding node and limited data at the target node to enhance the performance at the target node, as shown in Figure 1(b).

Under the transfer learning setting, there are three main challenges. First, transfer learning aims to exploit common and transferable knowledge in different data distributions. Netlist data consists of two kinds of knowledge: node-dependent and design-dependent. The former refers to the standard cell types and characteristics that vary across technology nodes, while the latter captures the functionalities of timing paths independent of the nodes. However, these two kinds of knowledge are highly intertwined in the netlist graph, making it difficult to leverage the common and transferable parts across different nodes. Second, the arrival time values of different timing paths can vary dramatically, even by one or two orders of magnitude, which poses significant challenges for the ML-based regression model. Third, the limited target node data makes the timing predictor susceptible to overfitting the training designs, which hinders the broad application of the learned model. To mitigate these challenges, we propose a novel framework tailored for timing prediction that first disentangles the features of the timing path into node-dependent and design-dependent parts in the latent feature space, then, we align these two parts separately by minimizing a node-based contrastive loss and a design-based discrepancy loss. Subsequently, with the disentangled features, we propose a new Bayesian machine learning-based framework that can adapt to highly variable arrival time values and generalize well to new designs.

To validate the effectiveness of our proposed method, we collect abundant data at the 130*nm* node and limited data at the 7*nm* node as our training set, then test its performance using the 7*nm* data. The experimental results show that our method outperforms its counterparts by a significant margin. The main contribution can be summarized as follows:

- To the best of our knowledge, we are the first to investigate transfer learning from different technology nodes in timing prediction.
- A new feature disentanglement framework is proposed, which first decouples the timing path features into node-dependent and design-dependent parts and aligns them separately.
- To enable effective generalization to new designs, we propose a novel Bayesian machine learning-based timing predictor.
- The experimental results on transfer learning from 130*nm* node to 7*nm* node verify the effectiveness of our method.

2 Preliminaries

2.1 ML-based Timing Prediction

With the rapid development of machine learning, ML-based timing prediction techniques have been explored and proven to be successful. These approaches leverage large datasets and utilize ML algorithms to model the circuit timing. In [2], the authors propose a random forest model that first utilizes extracted placement features to predict routed net delay and slew. Then, relying on PERT traversals [5], the overall circuit timing is determined based on the prediction results output by the random forest model. To further accelerate the timing prediction, Guo *et al.* [3] propose a timing engine-inspired GNN model. By representing netlists as graphs, GNNs can capture the relationships and dependencies between circuit elements. The proposed GNN model predicts global timing metrics at timing endpoints in an end-to-end fashion, eliminating the need for additional feature engineering and invoking STA tools.

However, most previous ML-based timing predictors rely solely on the provided pre-routing netlist structures, which do not align with the optimized sign-off structures. Consequently, these methods often yield inaccurate timing prediction results. In order to address this issue, Wang *et al.* [4] introduce a timing optimization-aware predictor capable of handling netlist restructuring. By recognizing that timing endpoints remain unchanged during timing optimization, the authors focus on global endpoint-level prediction instead of the previously utilized local cell-level prediction. Besides, they propose incorporating supplementary information from the layout to model the impact of timing optimization. This study demonstrates that adopting a global endpoint-wise perspective from both the netlist and layout significantly enhances optimization-aware timing estimation. Following [4], our proposed timing model is also timing optimization-aware.

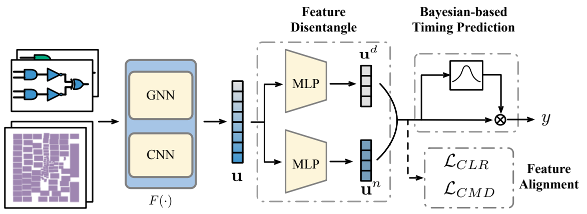

Figure 2: Overview of the method. Timing path feature alignment is only computed during the training stage.

2.2 Transfer Learning

In machine learning, when we do not have enough data on the target domain, an effective solution is transfer learning, which leverages data from other domains to aid the learning on the target domain. The critical issue in transfer learning is how to design a learning framework for learning the transferrable knowledge between the source and target domains. A widely used technique is pretraining-then-finetuning [6], which first trains the model on the source domain with abundant data to learn a good feature extractor and then finetunes the model with much fewer steps on the target domain to transfer the knowledge. Another effective approach is the parameter-sharing strategy [7] that lets some parameters of the neural networks be shared by data from different domains while the other parameters are learned separately. In EDA, [8] proposes a transfer learning-based framework for transistor sizing that transfers knowledge in different circuits. In the case of timing prediction, collecting the timing data requires running a long time-consuming toolchain. Consequently, the ability to transfer knowledge from one technology node to another becomes increasingly important. To the best of our knowledge, the application of transfer learning in the field of timing prediction has not been previously discussed.

2.3 Problem Formulation

Problem 1 (Transferrable Timing Prediction)**.** Given a large netlist set $\mathcal{N}_S$ at the source preceding technology node and a limited netlist set $\mathcal{N}_T$ at the target advanced technology node, our goal is to learn a model which accurately predicts endpoint arrival time on test netlist data at advanced technology node, achieving high R^2 score and demanding low computation cost.

3 Algorithm

We build our timing prediction model in an endpoint-based manner. The main idea is to disentangle the node-dependent and design-dependent features for each timing path. Then, we try to make the node-dependent features consistent in one technology node and distinguishable in different nodes. Meanwhile, since the design-dependent features are node-agnostic, we try to minimize the gap between the design-dependent features in different nodes. These two objectives serve as the target of feature alignment.

3.1 Timing Path Feature Extractor

Following [4], we build a timing path feature extractor $F(\cdot)$ that is capable of handling netlist restructuring, as shown in Figure 3. Given any netlist $\mathcal{G}$, we denote the set of all the timing paths as $\mathrm{Path}(\mathcal{G}) = \{\mathcal{G}'_i\}_{i=1}^{M}$, where $\mathcal{G}'$ represents a timing path and M is the total number of timing paths. Here $\mathcal{G}'$ is the whole fanin cone for the timing endpoint, a sub-graph of the netlist $\mathcal{G}$. For each timing path $\mathcal{G}'$, we can collect two types of input: the graph consisting of

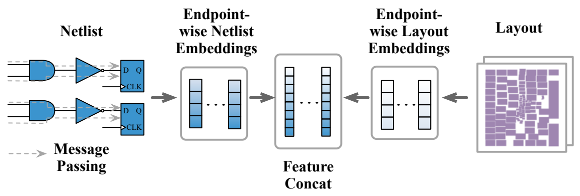
Netlist
Endpoint-wise Netlist Embeddings
Endpoint-wise Layout Embeddings
Layout
Message Passing
Feature Concat

Figure 3: Timing path feature extractor.

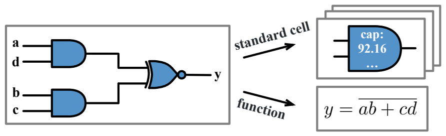

Figure 4: A netlist can be characterized by two parts of information: the functionality and the standard cell information.

pin and net information $\mathcal{H}$ and the corresponding layout image set $\mathcal{X}$.

Specifically, the layout image set $\mathcal{X}$ includes the cell density map, rectangular uniform wire density map, and the macro cells region map. Besides, $\mathcal{H}$ is a heterogeneous graph with two types of edges: the net edge connecting a net's drive pin and one of its sink pins, and the cell edge connecting one of a cell's input pins and its output pin. The nodes in $\mathcal{H}$ represent the pins in the netlist. The net distance, cell driving strength, gate type, and pin capacitance are used as the node features. Note that we use one-hot representation for the gate type and merge all the gates in different technology nodes as the total gate set. Then, we use a GNN to propagate on $\mathcal{H}$ from the primary inputs to the endpoints to obtain the feature for each timing path $\mathcal{G}'$. Meanwhile, we mask each timing path with the pin locations on the layout image and use a convolution neural network (CNN) to extract the features of the timing path there. The final feature of a given timing path is the concatenation of these two parts, which can be denoted as:

$$\mathbf{u} = F(\mathcal{G}') = [\mathrm{GNN}(\mathcal{H}), \mathrm{CNN}(\mathcal{X})] \in \mathbb{R}^m. \tag{1}$$

3.2 Timing Path Feature Disentanglement

As shown in Figure 4, each netlist contains two parts of information: the functionality information encoded in the design specification and the standard cell information, including its structures and parameters, such as the load and capacitance. For a given design, mapping to different technology nodes may produce two utterly different netlist graphs, but they will share the same logical functionality. Inspired by this, we aim to separate these two kinds of knowledge to facilitate transfer learning. However, these two pieces of knowledge are highly coupled in the netlist graphs, which is infeasible to separate directly from the raw input. Therefore, we propose disentangling the design-dependent knowledge and the node-dependent knowledge from the feature space of the timing path.

Since we aim to learn a performant timing predictor from only limited data in the target advanced technology node and abundant data in the source preceding technology node, we may assume that our training set is composed of two parts $\mathcal{N}_S = \{\mathcal{G}_1^S, \cdots, \mathcal{G}_{L_S}^S\}$ and $\mathcal{N}_T = \{\mathcal{G}_1^T, \cdots, \mathcal{G}_{L_T}^T\}$, where $\mathcal{N}_S$ and $\mathcal{N}_T$ represent the source preceding node and target advanced technology node netlist set, respectively, and we have $L_S \gg L_T$. For any path feature $\mathbf{u}$, we further

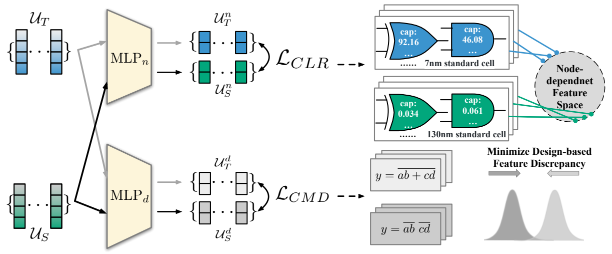

Figure 5: Timing path feature alignment.

adopt two multi-layer perceptrons (MLP$(\cdot)$) to disentangle the equal-sized node-dependent features $\mathbf{u}^n$ and design-dependent features $\mathbf{u}^d$ by:

$$\mathbf{u}^n = \mathrm{MLP}_n(\mathbf{u}) \in \mathbb{R}^{m/2}, \quad \mathbf{u}^d = \mathrm{MLP}_d(\mathbf{u}) \in \mathbb{R}^{m/2}. \tag{2}$$

For MLP$_n$, we use two linear layers and one ReLU activation in between. For MLP$_d$, we append one extra tanh activation after the MLP to limit the range of the design-dependent feature, which will be used for feature alignment detailed in the following section.

3.3 Timing Path Feature Alignment

Following feature disentanglement, the target of feature alignment is to design a proper loss function to encode our designated disentanglement of the node- and design-dependent knowledge into an end-to-end machine learning framework. With this in mind, we propose two loss functions, node-based contrastive loss and design-based discrepancy loss, to align the node- and design-dependent features, respectively. The overview of timing path feature alignment is shown in Figure 5.

Node-based contrastive loss. The intuition to align the node-dependent features is that the netlist on the same node should share the same standard cells, including the gate structures and the configuration parameters for pins and nets. On the contrary, the node-dependent features should be distinguishable for netlists at different nodes. For this purpose, we propose a node-based contrastive loss.

Specifically, in each batch of training data, we first sample some designs $\mathcal{N}'_S \subseteq \mathcal{N}_S$ and $\mathcal{N}'_T \subseteq \mathcal{N}_T$. Then, following Equation (1) and Equation (2), we can obtain the disentangled path feature sets, i.e., node- and design-dependent features, for the source preceding node, denoted as $\mathcal{U}_S^n$ and $\mathcal{U}_S^d$. Similarly, we can obtain the node-dependent path features and design-dependent path features of the target advanced node, denoted as $\mathcal{U}_T^n$ and $\mathcal{U}_T^d$.

Denote the set of all the node-dependent features as $\mathcal{A} = \mathcal{U}_S^n \cup \mathcal{U}_T^n$. Given any node-dependent feature set $\mathcal{U}^n$ ($\mathcal{U}_S^n$ or $\mathcal{U}_T^n$), the contrastive loss for the feature set can be defined as:

$$\mathcal{L}_{Set}(\mathcal{U}^n) = \sum_{\mathbf{u} \in \mathcal{U}^n} \frac{-1}{|\mathcal{U}^n| - 1} \sum_{\mathbf{m} \in \mathcal{U}^n \setminus \{\mathbf{u}\}} \frac{\exp(\mathbf{u} \cdot \mathbf{m}/\tau)}{\sum_{\mathbf{a} \in \mathcal{A} \setminus \{\mathbf{u}\}} \exp(\mathbf{u} \cdot \mathbf{a}/\tau)}, \tag{3}$$

and the total contrastive loss can be formulated as:

$$\mathcal{L}_{CLR} = \frac{1}{|\mathcal{U}_S^n|} \mathcal{L}_{Set}(\mathcal{U}_S^n) + \frac{1}{|\mathcal{U}_T^n|} \mathcal{L}_{Set}(\mathcal{U}_T^n). \tag{4}$$

Node-based contrastive loss aims to pull together the features from the same node while pushing apart those from different nodes. With this loss, the node-dependent features on the same node will be approximately consistent and differ in different nodes.

Design-based discrepancy loss. The design-dependent features represent the abstract logical functionality of each netlist. For each design, we can opt for different technology nodes to synthesize

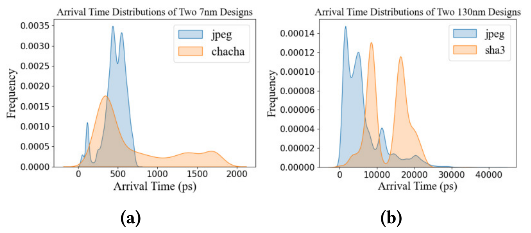

Figure 6: Kernel density estimation of the arrival time.

various netlists but with the same functionality. Therefore, the overall distribution of the design-dependent features in different nodes should be consistent.

To align the design-dependent features, we optimize the Central Moment Discrepancy (CMD) between the feature sets from different nodes, which can be formulated as:

$$\mathcal{L}_{CMD}(\mathcal{U}_S^d, \mathcal{U}_T^d) = \frac{1}{b-a}\left\|\mathbb{E}(\mathcal{U}_S^d) - \mathbb{E}(\mathcal{U}_T^d)\right\| + \sum_{k=2}^{\infty}\frac{1}{|b-a|^k}\left\|c_k(\mathcal{U}_S^d) - c_k(\mathcal{U}_T^d)\right\|, \tag{5}$$

where $[a, b]$ is the interval that bounds $\mathcal{U}_S^d$ and $\mathcal{U}_T^d$, $\mathbb{E}(\cdot)$ denotes the expectation and $c_k(\cdot)$ is the k-th order moment. Since we apply a tanh activation function after MLP$_d$, we limit the value of the node-dependent features in $(-1, 1)$. Therefore, we can set a and b to -1 and 1, respectively. In practice, we set the maximum moment order to 5. In essence, minimizing Equation (5) equals minimizing the gap between the statistics in different orders of the design-dependent features from different nodes. According to [9], we have the following properties:

Theorem 1. Let $\mathcal{U}_S^d$ and $\mathcal{U}_T^d$ be two probability distributions on a compact interval, then we have:

$$CMD(\mathcal{U}_S^d, \mathcal{U}_T^d) \to 0 \implies \mathcal{U}_S^d \to \mathcal{U}_T^d \tag{6}$$

Such a property can guarantee that minimizing the loss function formulated in Equation (5) can align the overall distribution of the design-dependent features from netlists at different technology nodes.

3.4 Bayesian-based timing prediction

Given a timing path feature $\mathbf{u}$, the conventional approach is to feed the timing path feature $\mathbf{u}$ into a linear layer with weight $\mathbf{W} \in \mathbb{R}^{1\times m}$ to output the arrival time. However, the value of the arrival time for different endpoints can be significantly different even in one netlist, as shown in Figure 6, which poses a huge challenge for accurate timing prediction with only a fixed $\mathbf{W}$. Besides, since we only have very limited data on our target technology node, the ML-based model is prone to overfitting the training design. This can dramatically limit the timing prediction performance on the unknown test netlists, which may possess a large distribution gap with the training set as shown in Figure 6. To address these issues, we propose a Bayesian-based timing prediction model, as shown in Figure 7. Under the Bayesian ML framework, the model parameters are considered as a distribution instead of fixed parameters. This feature allows us to condition the model parameters on any inputs, allowing the model to adapt to different inputs easily. In our timing prediction task, we model the final readout linear layer $\mathbf{W}$ as a distribution to obtain

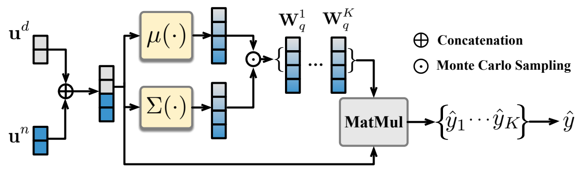

Figure 7: Bayesian machine learning-based timing prediction

better flexibility. Moreover, an ideal and well-generalizable timing predictor should make predictions based on the input timing path feature and the distribution of all the timing paths on the target node.

Therefore, following common practice in Bayesian ML [10, 11], we can formulate our learning objective as:

$$\log p(y|\mathcal{G}', \mathcal{N}) = \log \int p(y|\mathcal{G}', \mathbf{W})p(\mathbf{W}|\mathcal{N})d\mathbf{W}, \tag{7}$$

where y denotes the ground truth arrival time, $\mathcal{N}$ represents the overall distribution for all the timing paths at the technology node, and we condition $\mathbf{W}$ on this true global timing path distribution with a probability density function $p(\mathbf{W}|\mathcal{N})$, also known as the prior distribution. However, computing this probability in real practice is infeasible since the real timing path distribution of the whole target node $\mathcal{N}$ is intractable when we only have limited timing paths from the netlists on the target advanced technology node. Therefore, we adopt variational inference [10–12] to approximate the prior distribution.

Specifically, we introduce a variational posterior distribution $q(\mathbf{W}|\mathcal{G}')$ which only conditions on single timing path input and is easy to compute to simulate the prior distribution. Then, we can derive the evidence lower bound (ELBO) of the objective function by:

$$\begin{aligned}
\log p(y|\mathcal{G}', \mathcal{N}) &= \log \int p(y|\mathcal{G}', \mathbf{W})p(\mathbf{W}|\mathcal{N})d\mathbf{W} \\
&= \log \int p(y|\mathcal{G}', \mathbf{W})\frac{p(\mathbf{W}|\mathcal{N})}{q(\mathbf{W}|\mathcal{G}')}q(\mathbf{W}|\mathcal{G}')d\mathbf{W} \\
&= \log \mathbb{E}_q\left[p(y|\mathcal{G}', \mathbf{W})\frac{p(\mathbf{W}|\mathcal{N}))}{q(\mathbf{W}|\mathcal{G}')}\right] \\
&\geq \mathbb{E}_q\left[\log p(y|\mathcal{G}', \mathbf{W})\right] - \mathrm{KL}(q(\mathbf{W}|\mathcal{G}')||p(\mathbf{W}|\mathcal{N}))).
\end{aligned} \tag{8}$$

The first term in Equation (8) is the log-likelihood with the variational posterior, while the second term is the KL divergence between the variational posterior and the prior distribution. Equation (8) shows that we can learn a network with parameters only conditioned on the input timing path $\mathcal{G}'$ but with good generalization ability. Besides, this design also fits the feature of high variability of the arrival time of different timing paths.

Now, we introduce the construction of p and q for optimization. Following common practice in variational inference [10, 12], we can model the variational posterior distribution as a Gaussian formulated by:

$$\mathbf{W}_q \sim N(\mu([\mathbf{u}^n, \mathbf{u}^d]), \Sigma([\mathbf{u}^n, \mathbf{u}^d])), \tag{9}$$

where $\mathbf{W}_q$ is the linear layer generated by distribution q, $\mathbf{u}^n$ and $\mathbf{u}^d$ are the disentangled node-dependent and design-dependent features for the timing path $\mathcal{G}'$, and $\mu(\cdot)$ and $\Sigma(\cdot)$ are two MLPs. In other words, we use the two learnable small networks, $\mu(\cdot)$ and $\Sigma(\cdot)$, to output the mean and variance of the distribution of $\mathbf{W}_q$, as shown in

Table 1: Statistics of the dataset (edp stands for endpoint, e_n and e_c denote net edge and cell edge, respectively)

Benchmark		Input information				
		tech node	#pin	#edp	#e_n	#e_c
train	smallboom	7*nm*	694441	61764	488052	423344
	jpeg	130*nm*	1527166	39783	1150173	749294
	linkruncca	130*nm*	186546	17796	151617	84393
	spiMaster	130*nm*	99507	4739	75718	44917
	usbf_device	130*nm*	48104	4777	37557	21706
test	arm9	7*nm*	44469	2500	33065	29287
	chacha	7*nm*	35687	1986	25117	23083
	hwacha	7*nm*	1357798	61313	985057	922085
	or1200	7*nm*	1165114	172401	844443	658961
	sha3	7*nm*	794720	60323	552021	485596
Avg	train	7*nm*&130*nm*	511153	25772	380623	264731
	test	7*nm*	679558	59705	487941	423802

Figure 7. Similarly, using the amortization trick [12], we can model the prior as:

$$\mathbf{W}_p \sim N(\mu(\tilde{\mathbf{u}}(\mathcal{N})), \Sigma(\tilde{\mathbf{u}}(\mathcal{N}))), \tag{10}$$

where $\mathbf{W}_p$ is the linear layer generated by the prior distribution p, and $\tilde{\mathbf{u}}(\mathcal{N}) \in \mathbb{R}^m$ is a dummy timing path feature that represents the true distribution of all the timing paths within the whole technology node $\mathcal{N}$. To construct a representative $\tilde{\mathbf{u}}$ for the target technology node with only limited data, we leverage the disentangled node- and design-dependent features. Specifically, since the node-dependent features are consistent within one technology node, we can simply use the mean of all the node-dependent features to represent the node information. As for the design-related information, we can collect all the design-dependent features in both the source preceding technology node and the target advanced technology node since the design-based discrepancy loss has already brought them to the same distribution, and we can take the mean of them as the representative feature for all the designs.

Combining Equation (9), Equation (10) and the construction of $\tilde{\mathbf{u}}$, we can explicitly calculate $\mathrm{KL}(q(\mathbf{W}|\mathcal{G}')||p(\mathbf{W}|\mathcal{N}))$. To calculate the first term in Equation (8), we use Monte Carlo sampling to sample K samples from $q(\mathbf{W}|\mathcal{G}')$, denoted as $\{\mathbf{W}_q^i\}_{i=1}^K$, and with each sampled $\mathbf{W}_q^i$, we can obtain a timing prediction $\hat{y}_i$, which are then averaged to obtain the final timing prediction $\hat{y} = \frac{1}{K}\sum_1^K \hat{y}_i$. The ELBO objective can be formulated as:

$$\mathcal{L}_{ELBO}(y, \mathcal{G}', \mathcal{N}) = \frac{1}{K}\sum_{i=1}^{K} \log p(y|\mathcal{G}', \mathbf{W}_q^i) - \mathrm{KL}(q(\mathbf{W}|\mathcal{G}')||p(\mathbf{W}|\mathcal{N})). \tag{11}$$

Overall, the first term in Equation (11) represents the timing prediction loss, and the second term describes the distribution discrepancy between the variational posterior and prior distribution. Finally, the total loss is defined as:

$$\mathcal{L} = \sum_{\mathcal{N}\in\{\mathcal{N}_S,\mathcal{N}_T\}} \sum_{\mathcal{G}\in\mathcal{N}} \sum_{(y,\mathcal{G}')\in\mathrm{Path}(\mathcal{G})} \mathcal{L}_{ELBO}(y, \mathcal{G}', \mathcal{N}) + \gamma_1\mathcal{L}_{CLR} + \gamma_2\mathcal{L}_{CMD}, \tag{12}$$

where γ_1 and γ_2 are two hyper-parameters.

4 Experimental Results

4.1 Implementation Details

Our implementation is based on DGL [13] and Pytorch [14]. Training and evaluation are conducted on a Linux system with a 2.3GHz Intel Xeon CPU and a single NVIDIA GeForce RTX 3090 GPU. The hidden dimension is set to 256 fo the GNN. As for the CNN, the input size is $3 \times 512 \times 512$. Besides, the dimension of endpoint-wise netlist and layout embedding is set to 128. The weights for the node-based contrastive loss γ_1 and the designed-based discrepancy loss γ_2 are set to 10 and 100, respectively. Our model is trained with a learning rate of 0.0001 and batch size of 2048 for 200 epochs.

We collect netlists from two different technology nodes, 7*nm* and 130*nm*. We use four 130*nm* netlists and one 7*nm* netlist as the training set, and five 7*nm* netlists as the test set. The statistics of all the designs are shown in Table 1. Specifically, we collect open-source designs from Freecores [15] and Chipyard [16]. In the dataset generation workflow, we employ Cadence Genus with 130*nm* SkyWater [17] PDK for synthesis and Cadence Innovus for placement, timing optimization, routing and static timing analysis. We follow [4] to process the 7*nm* netlists. Timing constraints for each design in the PnR phase are derived from estimated values provided by Cadence Genus during synthesis to ensure effective timing optimization. DEF files and netlists are acquired at each phase of the flow, enabling the retrieval of pin and cell locations and other available features in accordance with PDK rules.

To verify the effectiveness of our proposed learning strategy, we set the following baselines for comparison. By default, all the baseline models are the previous state-of-the-art (SOTA) model published in DAC23 [4], but trained with different strategies. (1) The first baseline is only trained with limited advanced 7*nm* node netlist data, denoted as **DAC23-AdvOnly**. (2) The second baseline is simply merging the 7*nm* and the 130*nm* netlist data as the training set, denoted as **DAC23-SimpleMerge**. (3) The third baseline is parameter sharing [7], which is a common practice in transfer learning and multi-task learning, denoted as **DAC23-ParamShare**. For the 130*nm* and 7*nm* data, they share the same feature extractor, but adopt a node-specific linear layer for the final prediction. (4) The fourth baseline is pretraining-then-finetuning [6], denoted as **DAC23-PT-FT**. As a widely used technique in transfer learning, this strategy first trains the timing prediction model with 130*nm* netlist data and then fine-tunes it with 7*nm* netlist data.

4.2 Main Results

The main results are shown in Table 2. First of all, we can observe that only training with limited 7nm data (DAC23-AdvOnly) achieves poor performance, indicating the necessity of massive training data belonging to the same distribution as the test data.

The rest of the baselines investigate different approaches to leveraging the data from the 130*nm* node to help the learning on the 7*nm* node. The most intuitive one is to merge the limited 7*nm* and abundant 130*nm* data directly (DAC23-SimpleMerge). However, since the arrival time on 7*nm* and 130*nm* suffer from a large distribution discrepancy, as shown in Figure 6, this strategy is infeasible to handle them simultaneously, leading to a negative R^2 score. The next baseline parameter sharing (DAC23-ParamShare), as a common practice in transfer learning, obtains some improvements compared with DAC23-AdvOnly, demonstrating that it somehow leverages the knowledge from the 130*nm* data to help the learning for 7*nm* data. Another common practice in transfer learning, pretraining-then-finetuning (DAC23-PT-FT), achieves superior regression performance compared with DAC23-ParamShare with a substantial margin, indicating its effective knowledge transfer capability.

Table 2: The evaluation results on 7*nm* netlist data.

Baseline	DAC23 [4]-AdvOnly		DAC23 [4]-SimpleMerge		DAC23 [4]-ParamShare [7]		DAC23 [4]-PT-FT [6]		Ours	
	R^2 score	runtime	R^2 score	runtime	R^2 score	runtime	R^2 score	runtime	R^2 score	runtime
arm9	0.603	2.546	-2.069	2.546	0.567	2.546	0.837	2.546	**0.864**	2.621
chacha	0.624	1.188	-1.983	1.188	0.568	1.188	0.726	1.188	**0.890**	1.234
hwacha	0.170	5.229	-2.203	5.229	0.499	5.229	0.818	5.229	**0.828**	5.400
or1200	0.156	14.257	-6.037	14.257	0.240	14.257	0.209	14.257	**0.682**	14.793
sha3	0.425	1.690	-4.741	1.690	0.195	14.257	0.284	1.690	**0.785**	1.725
average	0.396	4.982	-3.407	4.982	0.414	4.982	0.575	4.982	**0.810**	5.154

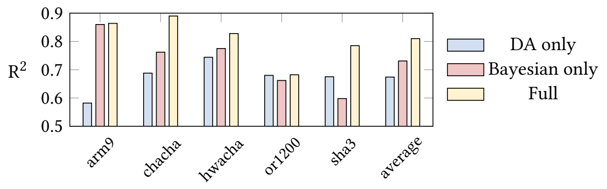

Figure 8: Ablation study on the effectiveness of each module.

Our method outperforms all the baselines with a significant margin (about 40% improvements on R^2 score compared with its best counterpart DAC23-PT-FT). This validates that our method is effective in handling the distribution shifts in the 130*nm* and 7*nm* data and is capable of transferring the knowledge to different technology nodes. Meanwhile, our method only takes around an extra 4% runtime to achieve this remarkable improvement due to the added parameters. Note that all the baseline models share the same runtime since they only differ in training strategy, and they are the same during inference.

4.3 Ablation Study

To further validate the effectiveness of our method, we conduct two ablation studies to show (1) the effectiveness of each module in our method and (2) the effect of the number of 130*nm* netlist data.

Effectiveness of each module.. To verify the effectiveness of the feature disentanglement and alignment (DA) and the Bayesian-based timing prediction module, we conduct experiments with only one of them, as shown in Figure 8. We can observe that, without any of them, the performance drops by a substantial margin, demonstrating the effectiveness of both of them. In addition, these two modules lead to different improvements on different designs. For instance, the model with DA only outperforms the model with Bayesian only on or1200 and sha3, but on arm9 and chacha model with Bayesian only owns substantial advantages.

The number of 130*nm* **netlist data.**. We also investigate how the number of 130*nm* data used in transfer learning affects the performance of the timing predictor. As shown in Table 3, when we increase the number of 130*nm* data, the timing prediction performance improves consistently. This indicates that, on the one hand, our method is effective in transferring the knowledge in different nodes to enhance the performance on the target node; on the other hand, the involvement of more 130*nm* data can improve the timing predictor's generalization ability on various 7*nm* designs.

5 Conclusion

ML-based methods [3, 4] have achieved remarkable success in pre-routing timing prediction. To ensure precise prediction, they demand extensive data from the designated technology node. However, the data collection process is time-consuming, posing a challenge in acquiring adequate data for advanced technology nodes. To mitigate this issue, we propose a novel transfer learning framework that leverages abundant data from preceding technology nodes to enhance learning on the target technology node. Specifically, our method first disentangles the timing path features into node- and design-dependent parts and aligns them separately. Then, we use a Bayesian machine learning-based model to predict the arrival time of each timing path, which can handle its high variability and generalize to new designs in the test set. Experimental results on transfer learning from 130*nm* node to 7*nm* node validate the effectiveness of our method.

Table 3: Ablation study on the number of 130*nm* **designs. J, L S, and U denote jpeg, linkruncca, spiMaster, and usbf_device, respectively. We report the** R^2 **score.**

J	L	S	U	arm9	chacha	hwacha	or1200	sha3	average
✓				0.496	0.394	0.649	0.363	0.631	0.507
✓	✓			0.312	0.773	0.470	0.616	0.599	0.554
✓	✓	✓		0.564	0.821	0.804	0.531	0.673	0.679
✓	✓	✓	✓	**0.864**	**0.890**	**0.828**	**0.682**	**0.785**	**0.810**

References

[1] J. Rubinstein, P. Penfield, and M. A. Horowitz, "Signal delay in rc tree networks," *IEEE TCAD*, vol. 2, no. 3, pp. 202–211, 1983.
[2] E. C. Barboza, N. Shukla, Y. Chen, and J. Hu, "Machine learning-based pre-routing timing prediction with reduced pessimism," in *Proc. DAC*, 2019, pp. 1–6.
[3] 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.
[4] Z. Wang, S. Liu, Y. Pu, S. Chen, T.-Y. Ho, and B. Yu, "Restructure-tolerant timing prediction via multimodal fusion," in *Proc. DAC*, 2023.
[5] H. Chang and S. S. Sapatnekar, "Statistical timing analysis considering spatial correlations using a single pert-like traversal," in *Proc. ICCAD*, 2003, pp. 621–625.
[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. NAACL*, 2019.
[7] Z. Zhang, P. Luo, C. C. Loy, and X. Tang, "Facial landmark detection by deep multi-task learning," in *Proc. ECCV*, 2014.
[8] H. Wang, K. Wang, J. Yang, L. Shen, N. Sun, H.-S. Lee, and S. Han, "GCN-RL circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning," in *Proc. DAC*, 2020.
[9] W. Zellinger, T. Grubinger, E. Lughofer, T. Natschläger, and S. Saminger-Platz, "Central moment discrepancy (cmd) for domain-invariant representation learning," *arXiv preprint arXiv:1702.08811*, 2017.
[10] Z. Xiao, X. Zhen, L. Shao, and C. G. Snoek, "Learning to generalize across domains on single test samples," in *Proc. ICLR*, 2022.
[11] J. Zhang, C. Zhao, B. Ni, M. Xu, and X. Yang, "Variational few-shot learning," in *Proc. ICCV*, 2019, pp. 1685–1694.
[12] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," in *Proc. ICLR*, 2014.
[13] M. Y. Wang, "Deep graph library: Towards efficient and scalable deep learning on graphs," in *ICLR workshop on representation learning on graphs and manifolds*, 2019.
[14] 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.
[15] "Freecores," https://github.com/freecores.
[16] "Chipyard," https://github.com/ucb-bar/chipyard.
[17] "Skywater Open Source PDK," https://github.com/google/skywater-pdk.