Recursive Learning-Based Virtual Buffering for Analytical Global Placement

Andrew B. Kahng, Yiting Liu, and Zhiang Wang
{abk,yil375,zhw033}@ucsd.edu

UC San Diego, La Jolla, CA, USA

***Abstract*—With scaling of interconnect versus gate delays in advanced technology nodes, placement with *buffer porosity* awareness is essential for timing closure in physical synthesis flows. However, existing approaches face two key challenges: (i) traditional van Ginneken-Lillis-style buffering approaches [20], [27] are computationally expensive during global placement; and (ii) machine learning-based approaches, such as *BufFormer* [18], omit important Electrical Rule Check (ERC) considerations and typically fail to "close the loop" back into the physical design flow. In this work, we propose *MLBuf-RePlAce*, an open-source learning-driven virtual buffering-aware analytical global placement framework, built on top of the OpenROAD infrastructure [34]. *MLBuf-RePlAce* adopts an efficient recursive learning-based generative buffering approach to predict buffer types and locations, addressing ERC violations during global placement. We compare *MLBuf-RePlAce* against the default virtual buffering-based timing-driven global placer in OpenROAD, using open-source testcases from the TILOS MacroPlacement [36] and OpenROAD-flow-scripts [33] repositories. Without degradation of post-route power, *MLBuf-RePlAce* achieves (maximum, average) improvements of (56%, 31%) in total negative slack (TNS) within the open-source OpenROAD flow. When evaluated by completion in a commercial flow, *MLBuf-RePlAce* achieves (maximum, average) improvements of (53%, 28%) in TNS with an average of 0.2% improvement in post-route power.**

***Index Terms*—Virtual Buffering, Placement, Generative Model**

I. Introduction

Global placement is a critical step in VLSI physical design. The quality and efficiency of global placement significantly impacts the timing closure of the place-and-route (P&R) flow. State-of-the-art analytical global placers [8] [11] [14] [22] typically adopt the electrostatics-based placement approach [23], formulating global placement as nonlinear programming under density constraints. GPU-accelerated placers [19] [17] [13] parallelize the computation of wirelength and density functions, achieving significant speedups. Additionally, for design implementation in advanced technology nodes, timing closure requires extensive buffer insertion [18] and brings a complex interplay with global placement. Placement with *buffer porosity* awareness [7] is needed to allocate enough space for buffer insertion at later optimization stages; typically, this requires invocation of a buffering engine multiple times during global placement.

Traditional van Ginneken-Lillis-style buffering approaches [20] [27] typically begin with constructing a Steiner minimum tree [9] or a timing-driven tree [4] [3], followed by a bottom-up dynamic programming process to determine buffer types and locations. However, the joint space of tree generation and buffer insertion can be huge [10], especially for nets with many sinks, making such approaches computationally expensive during global placement. Moreover, in a GPU-accelerated placer, actual buffer insertion changes the netlist topology, which requires updates to the netlist stored in GPU memory. Frequent memory accesses and updates significantly degrade the speed advantages provided by GPU acceleration. Therefore, it is essential to develop a **lightweight virtual buffering strategy** to guide the placer toward a buffer-porosity-aware solution without compromising computational efficiency.

Machine learning (ML)-based buffering approaches have recently attracted considerable attention in the research literature. Wu et al. [30] propose a reinforcement-learning (RL)-based approach for simultaneous gate sizing and buffer insertion. Liang et al. [18] introduce *BufFormer*, a generative machine learning framework targeting buffering at the Engineering Change Order (ECO) stage. While both approaches show promise in delay optimization, they share two limitations: (i) they lack a thorough consideration of Electrical Rule Check (ERC) (maximum slew, maximum capacitance and maximum fanout) constraints, and (ii) they do not "close the loop" back to evaluate in the full physical design flow.

In this work, we present a recursive learning-based virtual buffering framework for addressing ERC violations in VLSI placement. We further integrate the proposed generative ML framework into a high-quality open-source electrostatic-based global placer, enabling placement with buffer porosity awareness [3] [7]. Our main contributions are as follows.

- We propose *MLBuf*, a recursive learning-based generative buffering model that efficiently predicts buffer types and locations to address ERC violations. *MLBuf* employs a recursive strategy inspired by the bottom-up dynamic programming paradigm used in the classical van Ginneken-Lillis-style algorithms [20] [27]. Additionally, a differentiable clustering approach is proposed to avoid Steiner tree construction, enabling the exploration of more types of buffer-embedded trees.
- We develop *MLBuf-RePlAce*, a learning-driven virtual buffering-aware analytical global placement framework. To the best of our knowledge, we are the first to explore and assess ML-guided buffering for analytical placement in the context of the "full placement and optimization flow". *MLBuf-RePlAce* is built on the OpenROAD infrastructure with a permissive open-source license, enabling others to adapt it for future enhancements.
- We evaluate our *MLBuf-RePlAce* framework using both OpenROAD and commercial flows, along with open testcases from the TILOS MacroPlacement [36] and OpenROAD-flow-scripts [33] GitHub repositories. We compare our approach against the default virtual buffering-based timing-driven global placer in the OpenROAD flow. Without degradation of post-route power, our approach achieves (maximum, average) improvements of (56%, 31%) in total negative slack (TNS) when evaluated within the open-source OpenROAD flow. When evaluated by completion in a commercial flow, our approach achieves (maximum, average) improvements of (53%, 28%) in TNS.

The remaining sections are organized as follows. Section II introduces the terminology and background. Section III and Section IV discuss our approach. Section V shows experimental results, and Section VI concludes the paper.

II. Preliminaries

This section reviews the nonlinear analytical global placement method and presents the problem formulation for ML-based buffer insertion. All terms used in this paper are summarized in Table VI in Appendix A-A.

A. Nonlinear Analytical Global Placement

State-of-the-art global placers, such as *RePlAce* [8] and *DREAMPlace* [19], usually adopt the electrostatics-based placement ap-

979-8-3315-3762-3/25/$31.00

proach [23]. Let $V = \{v_1, v_2, ..., v_n\}$ represent cells and $E = \{e_1, e_2, ..., e_m\}$ represent nets. Let x_v denote the coordinates of the center of cell v. Electrostatics-based global placers dissect the placement region into a grid of non-overlapping bins and optimize wirelength under the density constraint:

$$\begin{aligned} &\min \quad \mathcal{W}(V, E; x_v) \\ &\text{s.t.} \quad A_{\text{cell}}(V; x_v) \leq A_{\text{grid}}, \quad \text{for each bin in the bin grid} \end{aligned} \tag{1}$$

where $\mathcal{W}(V, E; x_v)$ is the wirelength function, $A_{\text{cell}}(V; x_v)$ is the cell area function in bins, and A_{grid} is the total area allowed in a bin. In our proposed *MLBuf-RePlAce* framework, we dynamically adjust A_{grid} based on buffering results during the global placement process.

B. Problem Formulation

The net-level buffering problem during global placement is defined as follows. We are given:

- a driver pin, its location and input slew,[1]
- a set of sink pins and their locations,
- a buffer library with associated cell information, including area, input capacitance, output resistance and maximum output capacitance,
- the input capacitance of each sink,
- the output resistance of the driver,
- electrical properties of interconnects for estimating resistance and capacitance,
- electrical rule check (ERC) constraints for each cell (including buffers), i.e., maximum slew (max_slew), maximum capacitance (max_cap) and maximum fanout (max_fanout) constraints, and
- a maximum wirelength constraint (max_wirelength).

The objective is to construct a *buffer-embedded tree*, i.e., a tree with inserted buffer types and locations, that minimizes total buffer area while satisfying all ERC and maximum wirelength constraints.

III. Our approach

Figure 1 shows our approach. During timing-driven global placement, when the placement overflow reaches a threshold specified in a predefined overflow_list, a static timing engine (OpenSTA [35] in this work) is invoked to identify problematic nets with ERC violations [39]. Problematic nets, which also include long nets, are fed to the pre-trained *MLBuf* model for repair of ERC violations. (Details of *MLBuf* are given in Section IV.) *MLBuf* generates the buffer-embedded tree (with buffer types and locations) for each problematic net. Then *virtual buffering* is performed by pre-allocating *virtual occupancy* within placement bins according to the generated buffer-embedded trees. This pre-allocation allows the global placer to reserve adequate space for buffer insertion during later place-and-route stages, thus mitigating routing congestion while improving post-route power and timing.

More specifically, the virtual occupancy is calculated as follows. Let $\mathcal{H} = \{h_1, \ldots, h_i, \ldots, h_n\}$ denote predicted bounding boxes of virtual buffers, and $\mathcal{B} = \{b_1, \ldots, b_j, \ldots, b_m\}$ denote bins in the placement region. The overlapping area of (h_i, b_j) is calculated as

$$\begin{aligned} A_{\text{overlap}}(h_i, b_j) = &\max(0, \min(h_i^{rx}, b_j^{rx}) - \max(h_i^{lx}, b_j^{lx})) \\ &\times \max(0, \min(h_i^{ry}, b_j^{ry}) - \max(h_i^{ly}, b_j^{ly})), \end{aligned} \tag{2}$$

where (h_i^{lx}, h_i^{ly}) and (h_i^{rx}, h_i^{ry}) denote the lower-left and upper-right coordinates of the bounding box for the predicted buffer h_i, and similarly for bin b_j. The total buffer-induced area consumption in bin b_j is then scaled by its target density D_j, and subtracted from the available bin area:

$$A_{\text{grid}}(b_j) \mathrel{-}= A_{\text{overlap}}(h_i, b_j) \cdot D_j. \tag{3}$$

[1]The input slew is provided by OpenSTA [35].

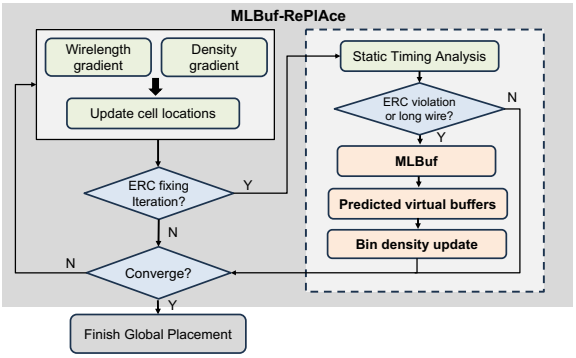

Fig. 1: The flow of *MLBuf-RePlAce*.

The resulting overflow for bin b_j is calculated as:

$$\text{overflow}(b_j) = \max\left(0, A'_{\text{cell}}(b_j) - A_{grid}(b_j)\right), \tag{4}$$

where $A'_{\text{cell}}(b_j)$ is the total area of movable cells placed in bin b_j. This formulation of virtual occupancy allows the global placer to account for the required buffer porosity, guiding cell distribution accordingly during global placement.

IV. MLBuf: Recursive Learning-based Buffering

This section introduces the recursive learning-based generative buffering model *MLBuf*. *MLBuf* mimics bottom-up dynamic programming to hierarchically construct the buffer-embedded tree for fixing ERC violations. We formulate the buffer insertion process as a sequence transformation problem. Each net is considered independently. Cells within the same net are regarded as a sequence with associated spatial and electrical information. *MLBuf* iteratively captures key features from input sequences to predict transformed sequences that include not only the original drivers and sinks, but also the inserted buffers. The output of each iteration is fed into the next iteration, enabling a hierarchical construction of buffer-embedded trees. Observe that no Steiner tree construction is needed when applying *MLBuf*.

Figure 2 illustrates *MLBuf*'s bottom-up learning process. In the first iteration (Figure 2(a1) and Figure 2(a2)), the input sequence is the original net with the driver v_{10} and all sinks $v_0, \ldots, v_5$. *MLBuf* determines which sinks need to be merged and driven by the same buffer, and output their corresponding "parent" buffers with buffer types and locations. In this example, sinks v_0 and v_1 are merged and driven by the buffer v_6; sink v_2 is driven by the buffer v_7; sinks v_3 and v_4 are merged and driven by the buffer v_8; and sink v_5 is not buffered, which is indicated by the corresponding buffer type None. As buffers shield the effects of downstream sinks, the input for the subsequent iteration (see the input in Figure 2(b2)) contains the driver (v_{10}), buffers predicted in the last iteration (v_6, v_7 and v_8), and any remaining sinks that were not buffered in the last iteration (v_5). The iterations end when all predicted buffer types are None, indicating no further buffering is needed (see the output in Figure 2(c2)). Then, the entire buffer-embedded tree is constructed (see Figure 2(c1)).

MLBuf introduces three key innovations, as follows.

First, constructing hierarchical buffer-embedded trees requires clustering of sinks to guide tree topology. Traditional clustering methods involve discrete operations that disrupt gradient propagation and prevent end-to-end optimization. To address this issue, we introduce a **differentiable clustering module** that enables end-to-end training, effectively bridging the gap between discrete clustering operations and continuous neural network optimization (see Section IV-A).

Second, the recursive structure of *MLBuf* can lead to error accumulation across iterations, potentially resulting in a buffer-

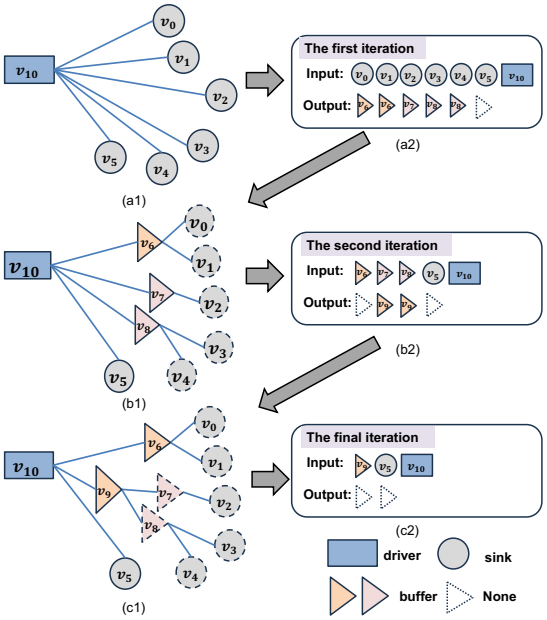

Fig. 2: *MLBuf*'s bottom-up learning process.

embedded tree with low accuracy. To mitigate this issue, we adopt the **teacher forcing strategy** [29] during training, which helps stabilize the learning process and reduce error propagation across recursive iterations (see Section IV-B).

Third, to encourage *MLBuf* to generate superior buffering solutions rather than just replicating ground-truth, we propose a specialized training paradigm incorporating both **inner-loop loss functions** and **outer-loop global penalties**. This dual-level optimization guides the model to actively search for and converge toward improved buffer insertion solutions beyond the provided labels (see Section IV-C).

A. Model Structure

Figure 3 shows the structure of *MLBuf*. There are three key components: feature encoders, a differentiable clustering module, and two task-specific decoders. Feature encoders extract spatial and electrical features of cells within the same net to provide meaningful representations for downstream modules. The clustering module determines which sinks need to be merged (clustered). Then, *Type_decoder* predicts the buffer type for each cluster and *Loc_decoder* outputs the corresponding buffer locations.

Feature Encoders: The shared feature encoder, *Shared_encoder*, takes as input a feature matrix $F \in \mathbb{R}^{N\times d_f}$, where each row represents a d_f-dimensional feature vector of a cell. The input features are detailed in Appendix A-B. The output of the encoder is $\tilde{x}^a \in \mathbb{R}^{N\times d_a}$, where d_a denotes the embedding dimension. To support task-specific learning, two additional encoders are used for feature augmentation. *Loc_encoder* processes spatial features (coordinates and distances) to produce spatial embeddings $\tilde{x}^l \in \mathbb{R}^{N\times d_l}$. *Elc_encoder* encodes electrical characteristics including the input/output slew, input/output capacitance, and max capacitance, yielding electrical embeddings $\tilde{x}^t \in \mathbb{R}^{N\times d_t}$. All encoders employ the self-attention mechanism to model the correlations among cells within the same net. The resulting embeddings $\tilde{x}^a$, $\tilde{x}^l$ and $\tilde{x}^t$ are forwarded to downstream modules for subsequent tasks.

Clustering Module: To avoid expensive Steiner tree computations, *MLBuf* employs a differentiable clustering module that dynamically groups sinks into clusters for hierarchical buffer-embedded tree construction. Sinks in the same cluster are jointly considered to determine whether the cluster needs to be buffered. The driver remains separate from all clusters and serves as the root of the tree. As the spatial embedding $\tilde{x}^l$ is important for clustering, the module takes as input the concatenated embedding $\tilde{x}^a \oplus \tilde{x}^l$ and computes a new representation $\tilde{x}^u$ via a three-layer fully-connected network (FCN) f_{θ_c}:

$$\tilde{x}^u = f_{\theta_c}[\tilde{x}^a \oplus \tilde{x}^l] \in \mathbb{R}^{N\times d_u}. \tag{5}$$

To enable end-to-end training, a soft cluster assignment matrix $M_k \in \mathbb{R}^{N\times K}$ is generated using Gumbel-Softmax [16],

$$[M_k]ij = \frac{\exp\left((f_{\theta_{c2}}(\tilde{x}^u)_{ij} + g_{ij})/\tau\right)}{\sum_{q=1}^{K}\exp\left((f_{\theta_{c2}}(\tilde{x}^u)_{iq} + g_{iq})/\tau\right)}, \tag{6}$$

where $g_{ij} \sim \text{Gumbel}(0,1)$ is Gumbel noise, and τ is a temperature parameter that controls the "sharpness" of the distribution. As $\tau \to 0$, the distribution becomes closer to a one-hot vector. This relaxation allows the model to learn cluster assignments in a fully differentiable manner. The resulting matrix M_k serves as a soft indicator of cluster membership: each row i corresponds to a cell, and each column j represents the probabilities of cells being assigned to cluster $j \in K$. We utilize M_k bidirectionally: (i) to aggregate cell-level features into cluster-level representations via weighted aggregation (Eq. 7), and (ii) to project cluster-level predictions back to individual cells (Eq. 10).

To this end, cluster-level representations are computed as

$$\begin{aligned}
\tilde{x}_c^u &= M_k^T \tilde{x}^u \in \mathbb{R}^{K\times d_u},\\
\tilde{x}_c^{al} &= M_k^T[\tilde{x}^a \oplus \tilde{x}^l] \in \mathbb{R}^{K\times d_{a+l}},\\
\tilde{x}_c^{at} &= M_k^T[\tilde{x}^a \oplus \tilde{x}^t] \in \mathbb{R}^{K\times d_{a+t}},
\end{aligned} \tag{7}$$

where $\tilde{x}_c^u$, $\tilde{x}_c^{al}$ and $\tilde{x}_c^{at}$ denote cluster-level embeddings. These embeddings are fed into decoders for predicting buffer types and locations at the cluster level.

Decoders: Two task-specific decoders based on self-attention are employed to predict buffer types and locations. The buffer type decoder, *Type_decoder*, predicts buffer types $Z_c^{type} \in \mathbb{R}^{K\times(X+1)}$ for each cluster:

$$Z_c^{type} = f_{\theta_{type}}(t_{\zeta_{type}}[\tilde{x}_c^u \oplus \tilde{x}_c^{al}]) \in \mathbb{R}^{K\times(X+1)}, \tag{8}$$

where $f_{\theta_{type}}$ represents the FCN with learnable parameter θ_{type}, and $t_{\zeta_{type}}$ represents the self-attention layers with learnable parameter ζ_{type}. The model considers X buffer types, along with a special "None" class represented by a zero vector in the one-hot encoding scheme, indicating that no buffer is needed. As the buffer type influences the buffer location, the predicted buffer types are also provided as an input to the location decoder.

The location decoder, *Loc_decoder*, predicts buffer coordinates $Z_c^{loc} \in \mathbb{R}^{K\times 2}$ for each cluster as

$$Z_c^{loc} = f_{\theta_{loc}}(t_{\zeta_{loc}}[\tilde{x}_c^u \oplus \tilde{x}_c^{at} \oplus Z_c^{type}]) \in \mathbb{R}^{K\times 2}, \tag{9}$$

where $f_{\theta_{loc}}$ is an FCN with learnable parameter θ_{loc}, and $t_{\zeta_{loc}}$ represents the self-attention layers with parameter ζ_{loc}. To decouple the tasks and improve robustness, buffer locations are predicted for all clusters, regardless of the buffer necessity predicted by *Type_decoder*. This approach mitigates potential error accumulation caused by inaccurate buffer type predictions.

Cell-level Mapping: Cluster-level predictions are mapped back to individual cells using the soft assignment matrix M_k [31]:

$$\begin{aligned}
Z^{type} &= M_k Z_c^{type} \in \mathbb{R}^{N\times(X+1)}\\
Z^{loc} &= M_k Z_c^{loc} \in \mathbb{R}^{N\times 2},
\end{aligned} \tag{10}$$

where Z^{type} and Z^{loc} are the assigned buffer type and location for each sink, derived from the correspoinding cluster-level predictions.

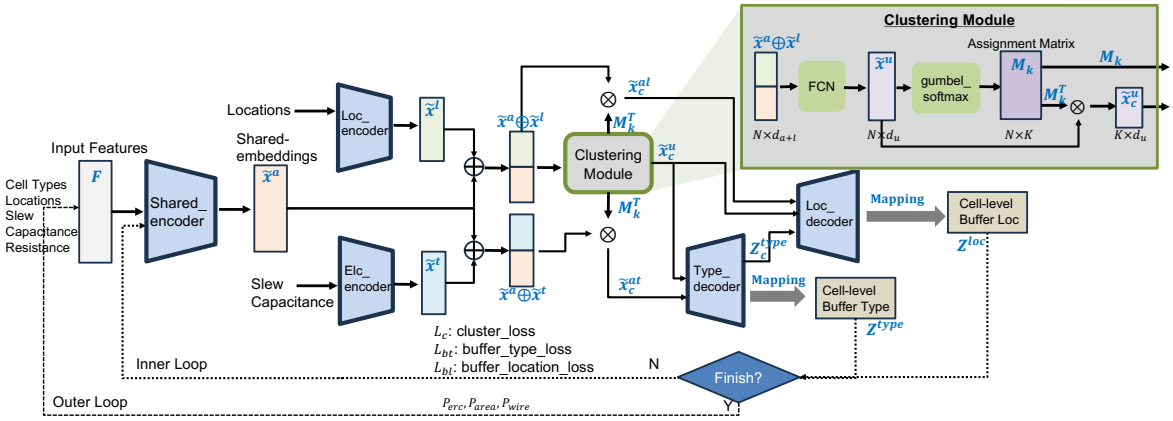

Fig. 3: Model structure of *MLBuf*.

B. Training Strategy

This section describes the training and inference strategy for *MLBuf*. We use buffer-embedded trees generated by OpenROAD Resizer (OR rsz) during global placement (*repairDesign()* in RepairDesign.cc [40]) as the ground truth. We define the *level* of a cell in the buffer-embedded tree as the longest path from itself to the sink. Suppose a buffer-embedded tree has $\tilde{H}$ levels (see Figure 2(c1)); it thus contains $\tilde{H}$ input-label pairs (see Figure 2(a2)(b2)(c2)) and can be constructed through $\tilde{H}$ iterations. More specifically, the input of the first iteration consists of the driver and all sinks. The corresponding label gives the sinks "parent" buffers in level 1 or None denoting no buffer. For the $i^{th}(1 \leq i \leq \tilde{H})$ iteration, its input consists of the driver, the buffers predicted in the $(i-1)^{st}$ iteration, and the sinks whose parent buffer is None in the $(i-1)^{st}$ iteration. Their respective labels are their parent buffers in the level i or None.

The training process contains two essential types of loop: *inner* loop and *outer* loop. The inner loop hierarchically builds the buffer-embedded tree by sequentially learning cluster assignments, buffer types, and buffer locations at each level of hierarchy. The outer loop evaluates the overall quality of the entire predicted buffered tree. To mitigate potential instability and error accumulation, we adopt *teacher forcing* [29] during training. Specifically, instead of using the model's prediction as input for the next iteration, we construct each training iteration using the ground-truth input-label pairs derived from the buffer-embedded tree. That is, we always feed the ground-truth input into the model. This approach provides accurate supervision at each iteration, improving training stability and enabling *MLBuf* to learn more reliable hierarchical representations. The pseudocode of MLBuf training is provided as Algorithm 1 in Appendix A-C.

During inference, *MLBuf* constructs the buffer-embedded tree in a fully auto-regressive and recursive manner. Predictions from each iteration are propagated to guide the next iteration, enabling sequential decision-making aligned with the tree structure. The process terminates when all predicted buffer types are zero, indicating that no further buffering is needed. To ensure accurate estimation of cell characteristics, we dynamically update cell features (slew and capacitance) across iterations. When new buffers are predicted during an iteration, we update the driver's output capacitance and output slew, as well as the input slew of newly inserted buffers and unbuffered sinks. The detailed calculation formulas for these updates are provided in Appendix A-B. The pseudocode of MLBuf inference is provided as Algorithm 2 in Appendix A-C.

C. Training Loss

The training loss includes both the loss functions in the inner loop and global-level penalties in the outer loop. They jointly guide the model optimization while enabling exploration of new buffer-embedded trees, rather than just following the given labels.

Inner-Loop Loss Functions: The inner loop constructs the buffer-embedded tree hierarchically, guided by three loss functions: cluster prediction loss L_c, buffer type classification loss L_{bt}, and buffer location regression loss L_{bl}.

The cluster loss L_c is a contrastive loss that encourages *MLBuf* to learn meaningful sink embeddings: it pulls together embeddings of sinks belonging to the same cluster in the ground truth and pushes apart those from different clusters. The formulation of the cluster loss is described in Appendix A-B. The buffer type loss L_{bt} is a multi-class classification loss, where each buffer type—including the "no buffer" case—is treated as a distinct class. Focal Loss [21] is applied to address the class imbalance issue. The buffer location loss L_{bl} is a mean squared error (MSE) loss between the predicted and ground-truth buffer coordinates.

Outer-Loop Global Penalties: After predicting all buffers, *MLBuf* has constructed a complete buffer-embedded tree. To further enhance the prediction quality and encourage the exploration of more advanced buffer-embedded tree structures, we incorporate three global penalty terms: ERC penalty P_{erc}, wirelength penalty P_{wire}, and buffer area penalty P_{area}.

P_{erc} penalizes ERC violations in the predicted buffer-embedded tree. It evaluates the output capacitance and fanout of drivers and predicted buffers, comparing them against their max capacitance $C_{max}(p)$ and max fanout $O_{max}(p)$ constraints. Maximum slew constraints are not explicitly enforced, as their effects are already closely aligned with output capacitance. P_{wire} penalizes wirelength violations. We use HPWL to estimate the wirelength between the driver (or a buffer) and its connected cells (i.e., direct fanouts), and impose penalties when this length exceeds the predefined threshold $\mathcal{W}_{max}$. P_{area} serves as a regularization term that encourages the model to minimize the total area of inserted buffers, thereby balancing ERC compliance with resource efficiency. Detailed formulations of these penalties are provided in Appendix A-B.

V. Experimental Validation

MLBuf is implemented using PyTorch Geometric. It has been integrated with RePlAce on the OpenROAD infrastructure to achieve virtual buffering during global placement. All codes and scripts are publicly released in the Github repository [38]. We run all experiments on a Linux server with an AMD EPYC 7742 64-Core Processor CPU (128 threads), 503GB RAM, and an NVIDIA A100-SXM4-80GB GPU. We evaluate *MLBuf* using five testcases

TABLE I: Characteristics of our benchmarks.

Design (NG45)	#Insts	#Nets	TCP_{OR}	TCP_{Invs}
ibex	16907	17728	2.20	-
jpeg	53042	58898	1.40	-
ariane	119256	142226	4.00	-
BlackParrot (BP)	768851	998716	NA	-
MegaBoom (MB)	1086920	1443755	NA	-

TABLE II: Characteristics of our dataset.

Characteristic	Training data	Validation data	Test data
Buffered tree count	174573	49878	24939
Sink count range	[1,725]	[1,678]	[1,680]
Buffer count range	[1,117]	[1,112]	[1,115]

(ibex, jpeg, ariane, BlackParrot (BP), and MegaBoom (MB)) in NanGate45 [37], which are publicly available in the OpenROAD [34] and MacroPlacement [36] GitHub repositories. Table I lists the statistical characteristics of these benchmarks. TCP_{OR} (ns) and TCP_{Invs} denote the target clock periods (TCP) used in the OpenROAD and Innovus[2] flows, respectively. TCP_{OR} is the default setting in OpenROAD-flow-scripts. TCP_{Invs} is specified by tuning such that post-route WNS falls within 5%–15% of the TCP.

We use five sizes of buffers from NanGate45, i.e., BUF_X2, BUF_X4, BUF_X8, BUF_X16 and BUF_X32. The training dataset is collected using the OpenROAD infrastructure. Specifically, we run the virtual buffering-based timing-driven global placer integrated in OpenROAD on publicly available designs (ibex, jpeg and ariane). When the placement overflow reaches a threshold specified in the predefined overflow_list,[3] buffer insertion is performed by OR rsz. We extract these buffered nets (i.e., buffer-embedded trees) for model training. The hyperparameter selection and training details of *MLBuf* are described in Appendix A-D1 and Appendix A-D2, respectively. The statistical characteristics of our collected dataset are summarized in Table II. Note that BP and MB are held out entirely during model training, and are used for unseen-design evaluation to evaluate the generalization of *MLBuf*. All collected data is publicly released in [38] to support future research.

In this section, we first present the evaluation of predicted buffers in Section V-A, and the effect of *MLBuf-RePlAce* on the full placement and optimization flow in Section V-B. We then provide the runtime breakdown of *MLBuf* in Section V-C. Finally, we analyze the benefits of buffer-porosity-aware placement with respect to end-to-end design quality in Section V-D.

A. Evaluation of Predicted Buffer-embedded Trees

We compare the performance of buffer-embedded trees predicted by *MLBuf* and those calculated by OR rsz. Specifically, during each timing optimization, we insert the predicted buffers back into netlists according to the topology of the buffer-embedded trees on ibex and jpeg. Then we compare the buffer area, and the number of ERC violations reported by OpenSTA. Note that timing optimization is performed a total of 13 times during the global placement process; in Table III, average values are used to represent performance. We can see that *MLBuf* achieves comparable ERC violation resolution to OR rsz. A detailed per-iteration comparison is provided in Appendix A-D3.

[2]We do not perform any benchmarking of commercial EDA tools. Further, to avoid inadvertent benchmarking, we mask the target clock period values TCP_{Invs} in Table I.

[3]By default, overflow_list = {0.7, 0.65, 0.6, 0.55, 0.5, 0.45, 0.4, 0.35, 0.3, 0.25, 0.2, 0.15, 0.1}. Thus, the timing engine is invoked 13 times throughout the global placement process.

TABLE III: Evaluation of predicted buffer-embedded trees.

Design	Method	Average results across 13 iterations				
		Buffer area	#Buffers	#ERC violations		
				#slew violations	#cap violations	#fanout violations
ibex	No_buf	0	0	77	174	0
	OR rsz	651	136	48	145	0
	MLBuf	684	82	58	154	0
jpeg	No_buf	0	0	4643	22	0
	OR rsz	1205	142	5	13	0
	MLBuf	386	37	13	19	0

B. PPA Validation

We now present post-route PPA results of *MLBuf-RePlAce* obtained using both the open-source OpenROAD flow [2] and the commercial flow. Since no existing ML-driven buffering method is integrated into analytical placement to complete the full "closed-loop" optimization flow, we evaluate the effectiveness of *MLBuf-RePlAce* (runscripts are in the Github repository [38]) by comparing it against the following three approaches:

- RePlAce: Pure RePlAce without timing-driven mode. This is the state-of-the-art open-source global placer in the OpenROAD project [34] (commit hash: df581be).
- TD-RePlAce: Default virtual buffering-based timing-driven global placement in OpenROAD. In this mode, OR rsz [40] is invoked during global placement to repair nets with ERC violations by inserting buffers. It updates net weights based on slack estimation and subsequently removes buffers to achieve virtual buffering (commit hash: df581be).
- Ad-Hoc Baseline: This is a rule-based analytical approximation used as a baseline for comparison. Detailed description of this approach is provided in Appendix A-D4. Note that all hyperparameters in this approach are tuned to enhance the accuracy of estimated buffer counts by benchmarking against OR rsz, thereby improving the competitiveness of this baseline.

We use the value of post-route wirelength, WNS, TNS, and power (with respect to TCP) for purposes of comparison.

PPA Validation with OpenROAD Flow. The flow begins with synthesis using Yosys, and floorplanning using OpenROAD. The synthesized netlist is debuffered, and then global placement is performed using RePlAce. During global placement, timing optimization is triggered when the placement overflow reaches each threshold specified in the predefined overflow_list. At each optimization iteration, problematic nets are identified through OpenSTA and fed into one of three virtual buffering strategies, *MLBuf*, OR rsz, or the Ad-Hoc baseline. When *MLBuf* or the Ad-Hoc baseline is employed, the bin densities are updated according to the predicted buffer types and locations (see Section III), enabling the placer to account for buffer area effects. When OR rsz is employed, we use the default virtual buffering strategy in OpenROAD, which adjusts net weights based on the estimated slacks. Subsequent steps correspond to OpenROAD's standard flow: resizing, buffering, legalization, detailed placement, clock tree synthesis (CTS), and routing. We compare the post-route PPA results in Table IV. We exclude ariane from the comparison, as all of its data collected from OpenROAD is used in training (see Appendix A-D2 for details). We also exclude BP and MB since OpenROAD fails to route for these designs. Compared with TD-RePlAce, *MLBuf-RePlAce* achieves up to 56% and an average of 31% improvement in TNS without power degradation.

PPA Validation with Commercial Flow. The flow begins with synthesis using Cadence Genus 21.1, a state-of-the-art commercial synthesis tool. The synthesized netlist is then debuffered, and floorplanning is conducted using Cadence Innovus 21.1. Global placement is subsequently performed using RePlAce within the OpenROAD framework, following the same methodology as described in the OpenROAD-based flow. During global placement, *MLBuf*, OR rsz or the Ad-Hoc baseline is invoked to perform virtual

TABLE IV: Post-route metrics evaluated with OpenROAD flow.

Design	Method	Metrics			
		rWL	WNS	TNS	Power
ibex	RePlAce	303209	-0.301	-22.750	128.661
	TD-RePlAce	299212	**-0.162**	-15.361	127.828
	Ad-Hoc	**298973**	-0.295	-30.389	128.876
	MLBuf	299049	-0.234	**-12.112**	**127.343**
jpeg	RePlAce	**630755**	-0.105	-4.669	**551.357**
	TD-RePlAce	634789	-0.146	-5.651	559.573
	Ad-Hoc	637868	-0.100	-3.611	573.129
	MLBuf	640555	**-0.082**	**-2.524**	567.337

TABLE V: Post-route metrics evaluated with Commercial flow (TCP values are masked).

Design	Method	Metrics				
		rWL	WNS	TNS	Power	#FEP
ibex	RePlAce	279617	-0.147	-137.747	44.640	1477
	TD-RePlAce	**277298**	**-0.140**	-111.615	44.948	1557
	Ad-Hoc	279687	-0.149	-127.012	44.940	1407
	MLBuf	281809	-0.143	**-103.690**	**44.528**	**1324**
jpeg	RePlAce	588917	-0.115	-85.659	536.972	2047
	TD-RePlAce	588099	-0.089	-68.505	547.551	1945
	Ad-Hoc	588906	-0.105	-82.612	537.902	2001
	MLBuf	**582866**	**-0.068**	**-46.013**	**535.932**	**1767**
ariane	RePlAce	4781145	-0.169	-157.028	846.221	2015
	TD-RePlAce	**4805300**	-0.134	-104.777	846.393	1865
	Ad-Hoc	4781035	-0.149	-126.201	846.112	1932
	MLBuf	4795461	**-0.106**	**-67.962**	**844.090**	**1562**
BP	RePlAce	**27306546**	-0.057	-34.223	4188.776	429
	TD-RePlAce	27373470	-0.067	-40.861	4194.181	504
	Ad-Hoc	27379181	-0.041	-35.763	4189.003	502
	MLBuf	27373073	**-0.038**	**-19.246**	**4188.754**	**421**
MB	RePlAce	41233170	-0.064	-1.777	2162.949	126
	TD-RePlAce	**39848996**	-0.028	-1.166	**2145.268**	92
	Ad-Hoc	40330039	-0.080	-2.070	2159.548	138
	MLBuf	41103476	**-0.026**	**-0.714**	2152.224	**72**

buffering. After global placement, the default optimization flow provided by Innovus is executed to complete the physical design and produce post-route PPA results. We compare post-route wirelength, WNS, TNS, power and number of failing endpoints (#FEP) in Table V. Compared with TD-RePlAce, *MLBuf-RePlAce* achieves (maximum, average) improvements of (24%, 17%) in WNS, (53%, 28%) in TNS, (2%, 0.2%) in post-route power, and (22%, 14%) in #FEP. In general, *MLBuf-RePlAce* consistently produces high-quality placement solutions, with better PPA outcomes compared to other methods.

C. Runtime Breakdown

Figure 4 provides a runtime comparison between OR rsz and *MLBuf*. OR rsz runs on a single CPU thread. The runtime contains the Steiner tree generation and buffer calculation (i.e., buffer type selection and buffer location calculation). For *MLBuf*, we assume that the features of the driver and sinks are already extracted, and we measure the runtime of generating an entire buffer-embedded tree (i.e., inference time). As net sizes increase, the runtime of OR rsz increases significantly. On the other hand, *MLBuf* consistently predicts buffer types and location with high efficiency, achieving over a 3× speedup on large nets compared to OR rsz. These results demonstrate the efficiency and scalability of *MLBuf*. Since each net can be considered independently during virtual buffering, our ongoing work further enhances the efficiency of *MLBuf-RePlAce* by parallelizing net processing.

D. Benefits of Buffer-porosity-aware Placement

This section analyzes the impact of buffer-porosity-aware placement on the full optimization flow, particularly under varying timing constraints. We systematically sweep the target clock period (TCP), from 0.6ns to 5.0ns, to adjust the stringency of timing requirements. To reduce noise, we perturb each TCP by ±10 ps and use the average TNS to represent performance. Decreasing the TCP tightens these timing constraints, and hence requires more buffer insertions for timing optimization.

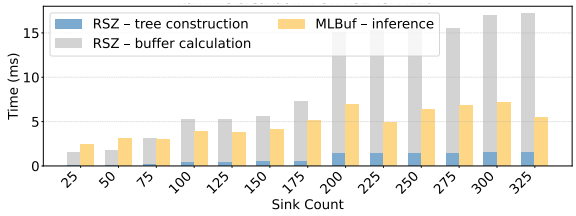

Fig. 4: Runtime comparison between MLBuf and OR rsz

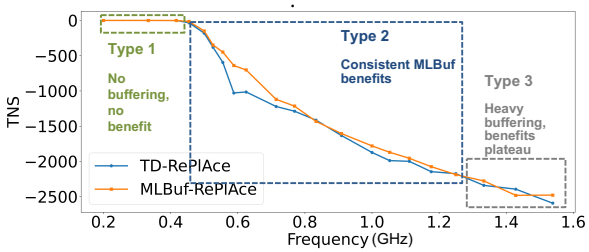

Fig. 5: Post-route TNS comparisons between *MLBuf-RePlAce* and TD-RePlAce under different clock frequencies.

Figure 5 shows the post-route TNS of *MLBuf-RePlAce* and TD-RePlAce across different clock frequencies (GHz) ($clock\_frequency = \frac{1}{TCP}$) on ibex under the OpenROAD infrastructure. We observe three *Types* of outcomes: (i) Type 1, where there is no buffering; (ii) Type 2, where *MLBuf* consistently outperforms TD-RePlAce; and (iii) Type 3, where benefits plateau with heavy buffering. In general, *MLBuf-RePlAce* delivers greater benefits with greater buffering needs, i.e., with Type 1 and Type 2 outcomes. *MLBuf* guides the placer to pre-allocate buffer space for downstream optimization, thereby reducing routing detours and achieving better PPA results. Our results also confirm the importance of buffer-porosity-aware global placer for the full optimization flow. However, as *MLBuf* is designed for resolving ERC violations and is insensitive to timing constraints, the improvements plateau when the TCP becomes too tight (Type 3), indicating that the number of buffers predicted by *MLBuf* is insufficient under extremely constrained timing. Our ongoing work further enhances *MLBuf* by incorporating timing-related features and constraints, enabling timing-aware virtual buffering prediction.

VI. Conclusion and Future Directions

We have presented *MLbuf* and *MLBuf-RePlAce*, a learning-driven virtual buffering-aware analytical global placement framework built upon the OpenROAD infrastructure. Experimental results demonstrate that *MLBuf-RePlAce* outperforms the default virtual buffering-based global placer in OpenROAD in terms of post-route WNS and TNS metrics, with no post-route power degradation. Ongoing extensions to *MLBuf-RePlAce* include: (i) enabling inverter-based repeater insertion and handling sink polarity, (ii) incorporating delay information into the virtual buffering model to enable net-weighted timing-driven global placement, and (iii) integrating the virtual buffering model with state-of-the-art GPU-accelerated global placers [17] [19] to support virtual buffering-aware timing-driven GPU-accelerated global placement. In combination with open-sourcing and OpenROAD integration, we believe that this work can strengthen foundations for future research in fast and high-quality timing-driven global placement.

Acknowledgments

This work is partially supported by the Samsung AI Center.

REFERENCES

[1] A. Agnesina, R. Liang, G. Pradipta, A. Rajaram and H. Ren, "GOALPlace: Begin with the End in Mind", *Proc. ISPD*, 2025, pp. 2-10.

[2] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem, G. Pradipta, S. Reda, M. Saligane, S. S. Sapatnekar, C. Sechen, M. Shalan, W. Swartz, L. Wang, Z. Wang, M. Woo and B. Xu, "Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project", *Proc. DAC*, 2019, pp. 1-4.

[3] C. J. Alpert, G. Gandham, M. Hrkic, Jiang Hu, S. T. Quay and C. N. Sze, "Porosity-Aware Buffered Steiner Tree Construction", *IEEE Trans. on CAD* 23(4) (2004), pp. 517-526.

[4] C. J. Alpert, M. Hrkic, J. Hu, A. B. Kahng, J. Lillis, B. Liu, S. T. Quay, S. S. Sapatnekar and A. J. Sullivan, "Buffered Steiner Trees for Difficult Instances", *Proc. ISPD*, 2021, pp. 4-9.

[5] C. J. Alpert, S. K. Karandikar, Z. Li, G.-J. Nam, S. T. Quay and H. Ren, "Techniques for Fast Physical Synthesis", *Proc. of the IEEE* 95(3) (2007), pp. 573-599.

[6] A. E. Caldwell, A. B. Kahng, S. Mantik, I. L. Markov and A. Zelikovsky, "On Wirelength Estimations for Row-Based Placement", *IEEE Trans. on CAD* 18(9) (1999), pp. 1265-1278.

[7] T.-C. Chen, A. Chakraborty and D. Z. Pan, "An Integrated Nonlinear Placement Framework with Congestion and Porosity Aware Buffer Planning", *Proc. DAC*, 2008, pp. 702-707.

[8] C.-K. Cheng, A. B. Kahng, I. Kang and L. Wang, "RePlAce: Advancing Solution Quality and Routability Validation in Global Placement", *IEEE Trans. on CAD* 38(9) (2019), pp. 1717-1730.

[9] C. Chu and Y. -C. Wong, "FLUTE: Fast Lookup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI Design", *IEEE Trans. on CAD* 27(1) (2008), pp. 70-83.

[10] J. Cong and X. Yuan, "Routing Tree Construction Under Fixed Buffer Locations", *Proc. DAC*, 2000, pp. 379–384.

[11] Y. Du, Z. Guo, Y. Lin, R. Wang and R. Huang, "Fusion of Global Placement and Gate Sizing with Differentiable Optimization", *Proc. ICCAD*, 2024, pp. 1-9.

[12] W. C. Elmore, "The Transient Response of Damped Linear Networks with Particular Regard to Wideband Amplifiers", *Journal of Applied Physics* 19(1) (1948), pp. 55-63.

[13] Z. Guo and Y. Lin, "Differentiable-Timing-Driven Global Placement", *Proc. DAC*, 2022, pp. 1315 - 1320.

[14] C. Guth, V. Livramento, R. Netto, R. Fonseca, J. L. Güntzel and L. Santos, "Timing-Driven Placement Based on Dynamic Net-Weighting for Efficient Slack Histogram Compression", *Proc. ISPD*, 2015, pp. 141–148.

[15] D. Hendrycks and K. Gimpel, "Gaussian Error Linear Units (GELUs)", *arXiv preprint arXiv:1606.08415*, 2016.

[16] E. Jang, S. Gu and B. Poole, "Categorical Reparameterization with Gumbel-Softmax", *Proc. ICLR*, 2017, pp. 1-12.

[17] A. B. Kahng and Z. Wang, "DG-RePlAce: A Dataflow-Driven GPU-Accelerated Analytical Global Placement Framework for Machine Learning Accelerators", *IEEE Trans. on CAD* 44(2) (2025), pp. 696-708.

[18] R. Liang, S. Nath, A. Rajaram, J. Hu, and H. Ren, "BufFormer: A Generative ML Framework for Scalable Buffering", *Proc. ASP-DAC*, 2023, pp. 264-270.

[19] P. Liao, S. Liu, Z. Chen, W. Lv, Y. Lin and B. Yu, "DREAMPlace 4.0: Timing-Driven Global Placement with Momentum-Based Net Weighting", *Proc. DATE*, 2022, pp. 939–944.

[20] J. Lillis, C.-K. Cheng and T.-T. Y. Lin, "Simultaneous Routing and Buffer Insertion for High Performance Interconnect", *Proc. GLSVLSI*, 1996, pp. 148-153.

[21] T.-Y. Lin, P. Goyal, R. Girshick, K. He and P. Dollar, "Focal Loss for Dense Object Detection", *IEEE Trans Pattern Anal Mach Intell* 42(2), 2020, pp. 318-327.

[22] Z. Lin, M. Wei, Y. Chen, P. Zou, J. Chen and Y.-W. Chang, "Electrostatics-Based Analytical Global Placement for Timing Optimization", *Proc. DATE*, 2024, pp. 1-6.

[23] J. Lu, P. Chen, C.-C. Chang, S. Lu, D. J.-H. Huang, C.-C Teng and C.-K. Cheng, "ePlace: Electrostatics-Based Placement Using Fast Fourier Transform and Nesterov's Method", *ACM Trans. Des. Autom. Electron. Syst.* 20(2) (2015), pp. 17:1-17:34

[24] L. Luo, Q. Zhou, Y. Cai, X. Hong and Y. Wang, "A Novel Technique Integrating Buffer Insertion into Timing Driven Placement", *Proc. ISCAS*, 2006, pp. 1-4.

[25] D. A. Papa, T. Luo, M. D. Moffitt, C. N. Sze, Z. Li and G.-J. Nam, "RUMBLE: An Incremental Timing-Driven Physical-Synthesis Optimization Algorithm", *IEEE Trans. on CAD* 27(12), 2008, pp. 2156-2168.

[26] A. Stefanidis, D. Mangiras, C. Nicopoulos, D. Chinnery and G. Dimitrakopoulos, "Autonomous Application of Netlist Transformations Inside Lagrangian Relaxation-Based Optimization", *IEEE Trans. on CAD* 40(8) (2021), pp. 1672-1686.

[27] L. P. P. P. Van Ginneken, "Buffer Placement in Distributed RC-tree Networks for Minimal Elmore Delay", *Proc. ISCAS*, 1990, pp. 865-868.

[28] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser and I. Polosukhin, "Attention Is All You Need", *Proc. NeurIPS*, 2017, pp. 6000-6010.

[29] R. J. Williams and D. Zipser, "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks", *Neural computation* 1(2) (1989), pp. 270–280.

[30] H. Wu, Z. Huang, X. Li and W. Zhu, "AiTO: Simultaneous gate sizing and buffer insertion for timing optimization with GNNs and RL", *Integration* 98 (2024), pp. 102211.

[31] R. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton and J. Leskovec, "Hierarchical Graph Representation Learning with Differentiable Pooling", *Proc. NeurIPS*, 2018, pp. 4805–4815.

[32] Gate Resizer. https://github.com/The-OpenROAD-Project/OpenROAD/tree/master/src/rsz

[33] OpenROAD-Flow-Scripts, *commit hash: 52d002*. https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts

[34] OpenROAD, *commit hash: df581be*. https://github.com/The-OpenROAD-Project/OpenROAD.

[35] OpenSTA. https://github.com/The-OpenROAD-Project/OpenSTA/tree/master.

[36] TILOS MacroPlacement repository. https://github.com/TILOS-AI-Institute/MacroPlacement

[37] NanGate45 PDK. https://eda.ncsu.edu/freepdk/freepdk45/

[38] The MLBuf repository. https://github.com/ABKGroup/MLBuf_MLCAD

[39] The global placement module in OpenROAD (gpl). https://github.com/The-OpenROAD-Project/OpenROAD/tree/master/src/gpl.

[40] The gate resizer module in openROAD (rsz). https://github.com/The-OpenROAD-Project/OpenROAD/tree/master/src/rsz.

APPENDIX A
SUPPLEMENTARY APPENDIX

A. Terminology and Notation

Table VI summarizes all terms and their definitions.

TABLE VI: Terminology and Notation.

Basic Entities	
v	Cell (driver, sink or buffer)
p	Cell pin or input-output pin
e	Net $e = \{p\}$
V	Set of all cells $\{v\}$
$\mathcal{V}^s$	Set of drivers and sinks
$\mathcal{V}^b$	Set of buffers
E	Set of all nets $\{e\}$
x_v	Cell location, $\forall v \in V$
N	Number of cells (driver and sinks) in one net
Electrical Properties	
$C_{in}(p)$	Input capacitance of pin p
$C_{out}(p)$	Output capacitance of pin p
$C_{max}(p)$	Max (load) capacitance of driver pin p
$S_{in}(p)$	Input slew of pin p
$S_{out}(p)$	Output slew of pin p
$O(p)$	Fanout of driver pin p
$O_{max}(p)$	Max fanout of driver pin p
R	Resistance
Placement and Geometry	
$\mathcal{W}$	Half-perimeter wirelength
$\mathcal{W}_{max}$	Max wirelength constraint
b	Rectangular bin in placement region
$\mathcal{B}$	Set of all bins $\{b\}$
h	Bounding box of the buffer
$\mathcal{H}$	Set of buffer bounding boxes $\{h\}$
A	Area of cells or bins
$A_{\text{grid}}(b_j)$	Total area allowed in bin b_j
A_{cell}	Cell area function in bins
A'_{cell}	Total area of movable cells placed in a bin
$A_{\text{overlap}}(h_i, b_j)$	Overlapping area of the buffer h_i and the bin b_j
D_j	Target density of bin b_j
H	Level of the buffer-embedded tree
Model Features and Embeddings	
$F \in \mathbb{R}^{n \times m}$	Feature matrix
$\tilde{x}$	Feature embedding
d	Feature dimension
K	Cluster number
M_k	Assignment matrix with k clusters
$\tilde{C}$	Cosine distance between embeddings
g	Gumbel noise
Z^{type}	Predicted buffer types
Z^{loc}	Predicted buffer locations
X	Number of buffer types
$\oplus$	Embedding concatenation
Learning and Training Terms	
L	Loss functions
P	Penalty terms
W	Weight of loss and penalty
f_θ	Fully connected networks with parameter θ
t_ζ	Self-attention layer with parameter ζ,
$\mathcal{F}_\delta$	MLBuf model with parameter δ, $\mathcal{F}_\delta = \text{Stack}(t_\zeta, f_\theta)$
α	Scaling factor for wirelength estimation
β	Fitting factor for output slew
$\{\mathbf{X}_i, \mathcal{Y}_i\}$	Input-label pairs in ground truth buffer-embedded tree
T	Ground truth buffer-embedded tree in training dataset
$\hat{T}$	Predicted buffer-embedded tree during model inference
$\mathcal{D}_{train}$	Training dataset

B. MLBuf Model Details

Input Feature List for MLBuf: The input features to the *Shared_encoder* (see Section IV-A) are listed as follows:

- 3-dimensional one-hot encoding representing the cell type (driver, sink or buffer);
- Relative coordinates of sinks (with the driver fixed at (0, 0)), measured in microns;
- Manhattan distance to the driver;
- Input slew (ps) of sinks (-1 for the driver);
- Output slew (ps) of the driver (-1 for sinks);
- Input capacitance (fF) of sinks (-1 for the driver);
- Output load capacitance (fF) of the driver (-1 for sinks);
- Maximum capacitance of the driver (-1 for sinks); and
- Resistance of the driver (-1 for sinks).

Feature Update Formulas During Inference: As described in Section IV-B, MLBuf dynamically updates the output load capacitance and slew of cells during model inference.

The output load capacitance of the driver is estimated as the sum of the wire capacitance $C(wire)$ and the input pin capacitances of its child cells $C_{in}(p_{child})$:

$$\begin{aligned} C_{out}(p_{drvr}) &= C(wire) + \sum C_{in}(p_{child}), \\ C(wire) &= \alpha\mathcal{W} \times C_{wire\_unit}, \end{aligned} \tag{11}$$

where $\mathcal{W}$ is the half-perimeter wirelength (HPWL) of the bounding box enclosing the driver and its children, α is a scaling factor, and C_{wire_unit} is the unit wire capacitance.

The output slew of the driver is estimated as

$$S_{out}(p_{drvr}) = (R_{drvr} + R(wire)) \times C_{out}(p_{drvr}) \times \beta, \tag{12}$$

where R_{drvr} is the driver's output resistance, R_{wire_unit} is the unit wire resistance, $R(wire) = \alpha\mathcal{W} \times R_{wire\_unit}$ is the resistance of a given wire, and β is a fitting factor that accounts for higher-order delay effects. The input slew of each child cell connected to the driver is estimated to be equal to the driver's output slew.

Cluster Loss Formulation: As described in Section IV-C, the cluster loss Lc is a contrastive loss that encourages *MLBuf* to learn meaningful sink embeddings. It is calculated as

$$L_c = -log[y \cdot \tilde{C}(\tilde{x}_i^u, \tilde{x}_j^u) + (1-y) \times (1 - \tilde{C}(\tilde{x}_i^u, \tilde{x}_j^u))], \tag{13}$$

where $\tilde{C}(\tilde{x}_i^u, \tilde{x}_j^u) = 0.5 \times \left(\frac{\tilde{x}_i^u \cdot \tilde{x}_j^u}{|\tilde{x}_i^u|_2 |\tilde{x}_j^u|_2} + 1\right)$ is the cosine similarity between the two embeddings, scaled to the range $[0, 1]$, and $y \in \{0, 1\}$ indicates whether the sink pair (i, j) belongs to the same cluster in the ground truth. When $y = 1$, the loss reduces to minimization of the similarity distance, effectively pulling the embeddings closer. When $y = 0$, the loss encourages separation, penalizing similarity between sinks that should not be clustered together.

Penalty Term Formulations: To guide *MLBuf* toward high-quality solutions, we introduce three global penalty terms as described in Section IV-C: the ERC penalty P_{erc}, the wirelength penalty P_{wire}, and the buffer area penalty P_{area}.

The ERC penalty P_{erc} penalizes ERC violations in the predicted buffer-embedded tree. It evaluates the output capacitance and fanout of the driver and predicted buffers, comparing them against their max capacitance $C_{max}(p)$ and max fanout $O_{max}(p)$ constraints as specified in the Liberty file. The output capacitance is estimated using Elmore delay (see Eq. (11)). Formally, the penalties are defined as

$$\begin{aligned} P_{cap} &= \text{ReLU}(C_{out}(p) - C_{max}(p)), \\ P_{fanout} &= \text{ReLU}(O(p) - O_{max}(p)) \end{aligned} \tag{14}$$

where $C_{out}(p)$ denotes the estimated output capacitance of the driver or buffer pin p, and $O(p)$ represents p's fanout. As noted earlier, maximum slew constraints are not explicitly enforced, as their effects are already closely aligned with output capacitance, and are implicitly captured by P_{cap}. We use the ReLU function to ensure that penalties are only activated when constraints are exceeded.

The wirelength penalty P_{wire} penalizes wirelength violations as

$$P_{wire} = \text{ReLU}(\alpha\mathcal{W} - \mathcal{W}_{max}), \tag{15}$$

where $\mathcal{W}$ denotes the HPWL of the bounding box enclosing the driver (or a buffer) and its fanouts, $\mathcal{W}_{max}$ is a predefined max wirelength, and α is a scaling coefficient.

Last, the buffer area penalty P_{area} is defined as

$$P_{area} = W_{area} \cdot \text{ReLU}(log(A_{total}) - log(A_{small})), \quad (16)$$

where A_{total} is the total area of inserted buffers, A_{small} is the area of the smallest buffer, and W_{area} is a weighting factor. The use of logarithmic scaling provides a smooth penalty gradient, with minimal penalties for small deviations and stronger regularization for larger areas.

C. Algorithm Flows of MLBuf Training and Inference

Algorithm 1 and Algorithm 2 respectively describe *MLBuf*'s training and inference procedures. More details can be found in Section IV-B.

***MLBuf* training flow (Algorithm 1).** Each ground truth buffer-embedded tree $T \in \mathcal{D}_{train}$ with $\tilde{H}$ levels is converted to $\tilde{H}$ input-label pairs $(\{\mathbf{X}_i\}_{i=1}^{\tilde{H}}, \{\mathcal{Y}_i\}_{i=1}^{\tilde{H}})$ (Lines 1-2). *MLBuf* is trained using teacher forcing strategy (Lines 4-13). Specifically, in the i^{th} training iteration ($i \in [1, \tilde{H}]$), *MLBuf* predicts cluster assignment matrix $\mathbf{M}_i$, buffer types $\mathbf{Z}_i^{type}$ and buffer locations $\mathbf{Z}_i^{loc}$ (Lines 5-7), and also calculates inner loss functions (Lines 8-11). The $(i+1)^{st}$ input-label pair from the ground truth is then used for the next training iteration. The recursive training ends once all $\tilde{H}$ iterations are completed. Finally, global-level penalties are calculated (Lines 14-15) and model parameters are updated via back-propagation (Line 16).

Algorithm 1 MLBuf Training Procedure

Input: Training set $\mathcal{D}_{\text{train}} = \{T_1, \ldots, T_N\}$, maximum tree depth $\tilde{H}$, learnable parameters δ, learning rate η
Output: Optimized model parameters $\delta^\star$

1: **for each** tree $T \in \mathcal{D}_{\text{train}}$ **do**
2: $(\{\mathbf{X}_i\}_{i=1}^{\tilde{H}}, \{\mathcal{Y}_i\}_{i=1}^{\tilde{H}}) \leftarrow$ BUILDPAIRS(T) ▷ {convert T into $\tilde{H}$ input–label pairs}
3: $L_{\text{inner}} \leftarrow 0$, $L_{\text{total}} \leftarrow 0$
4: **for** $i = 1$ **to** $\tilde{H}$ **do**
5: $\mathbf{M}_i \leftarrow \mathcal{F}_c(\mathbf{X}_i; \delta)$ ▷ {cluster assignment matrix}
6: $\mathbf{Z}_i^{\text{type}} \leftarrow \mathcal{F}_{\text{type}}(\mathbf{X}_i, \mathbf{M}_i; \delta)$ ▷ {buffer types}
7: $\mathbf{Z}_i^{\text{loc}} \leftarrow \mathcal{F}_{\text{loc}}(\mathbf{X}_i, \mathbf{M}_i, \mathbf{Z}_i^{\text{type}}; \delta)$ ▷ {buffer locations}
8: $L_c \leftarrow$ CONTRASTIVELOSS$(\mathbf{M}_i, \mathcal{Y}_i)$
9: $L_{bt} \leftarrow$ FOCALLOSS$(\mathbf{Z}_i^{\text{type}}, \mathcal{Y}_i)$
10: $L_{bl} \leftarrow$ MSELOSS$(\mathbf{Z}_i^{\text{loc}}, \mathcal{Y}_i)$
11: $L_{\text{inner}} \mathrel{+}= W_1^l L_c + W_2^l L_{bt} + W_3^l L_{bl}$
12: {*the (i+1)-th ground-truth pair is used automatically via* $\mathbf{X}_{i+1}, \mathcal{Y}_{i+1}$}
13: **end for**
14: $(P_{\text{erc}}, P_{\text{wire}}, P_{\text{area}}) \leftarrow$ GLOBALPENALTIES(T)
15: $L_{\text{total}} \leftarrow L_{\text{inner}} + W_1^p P_{\text{erc}} + W_2^p P_{\text{wire}} + W_3^p P_{\text{area}}$
16: $\delta \leftarrow \delta - \eta \nabla_\delta L_{\text{total}}$ ▷ {back-propagation}
17: **end for**

***MLBuf* inference flow (Algorithm 2).** Given a problematic net with driver and sinks (Line 3), we feed it into the pre-trained model $\mathcal{F}_{\delta^*}$ to predict cluster assignment, buffer types and buffer locations (Lines 4-6). The inference process terminates when all predicted buffer types are None (Lines 7-10). Otherwise, the features of the predicted buffers and the remaining unbuffered sinks are updated according to Eq. (11) and Eq. (12) (Line 11). These updated features are then used as input for the next inference iteration (Lines 12-14). Finally, the process outputs the complete buffer-embedded tree (Line 16).

D. Experimental Details

1) Hyperparameter Selection: The most important hyperparameter in *MLBuf* is the number of clusters k (i.e., the output dimension of the clustering module). It affects the structure of the predicted buffer-embedded tree. To determine the value of k, we vary k while keeping all other hyperparameters fixed, and evaluate the predicted buffer-embedded trees on the test dataset. Specifically, we insert the predicted buffers back into the netlist according to the predicted buffer-embedded tree with different values of k, and compare the numbers of ERC violations reported by OpenSTA as well as the total buffer areas.

Algorithm 2 MLBuf Inference Procedure

Input: Driver d and initial sink set $\mathcal{V}_0^s$, trained model $\mathcal{F}_{\delta^\star}$
Output: Predicted buffer–embedded tree $\hat{T}$

1: Initialize index $i \leftarrow 1$, buffer set $\mathcal{V}_0^b \leftarrow \emptyset$, tree $\hat{T} \leftarrow \emptyset$
2: **while true do**
3: $\mathbf{X}_i \leftarrow \{\mathcal{V}_{i-1}^b, \mathcal{V}_{i-1}^s\}$ ▷ {features of current buffers+sinks}
4: $\mathbf{M}_i \leftarrow \mathcal{F}_c(\mathbf{X}_i; \delta^\star)$ ▷ {cluster assignment}
5: $\mathbf{Z}_i^{\text{type}} \leftarrow \mathcal{F}_{\text{type}}(\mathbf{X}_i, \mathbf{M}_i; \delta^\star)$ ▷ {buffer types}
6: $\mathbf{Z}_i^{\text{loc}} \leftarrow \mathcal{F}_{\text{loc}}(\mathbf{X}_i, \mathbf{M}_i, \mathbf{Z}_i^{\text{type}}; \delta^\star)$ ▷ {buffer locations}
7: $\mathcal{V}_i^b \leftarrow$ EXTRACTBUFFERS$(\mathbf{Z}_i^{\text{type}}, \mathbf{Z}_i^{\text{loc}})$
8: **if** $\mathcal{V}_i^b = \emptyset$ **then**
9: **break** ▷ {all predicted types are NONE}
10: **end if**
11: Update electrical features of $\mathcal{V}_i^b$ and $\mathcal{V}_{i-1}^s$ using Eq. (11) and Eq. (12)
12: $\hat{T} \leftarrow \hat{T} \cup \mathcal{V}_i^b$ ▷ {merge new buffers into the tree}
13: $\mathcal{V}_i^s \leftarrow$ unbuffered sinks after inserting $\mathcal{V}_i^b$
14: $i \leftarrow i + 1$
15: **end while**
16: **return** $\hat{T}$

We normalize the number of ERC violations and the total buffer area to the results obtained using *MLBuf* with $k = 5$. Figure 6 (left) shows the normalized ERC violations across different k values during each timing optimization iteration. Figure 6 (right) presents the corresponding buffer areas predicted by *MLBuf*. We can see that $k = 20$ yields the best performance among all values evaluated. It addresses more ERC violations while using less buffer area. Consequently, we set $k = 20$ as the default in our model. The default settings of other hyperparameters are similarly determined, and are provided in Table VII.

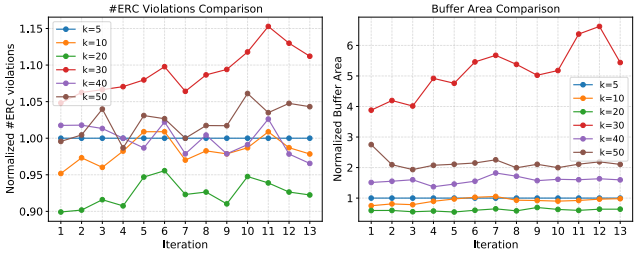

Fig. 6: Hyperparameter evaluation. The normalized number of ERC violations (left) and normalized buffer area (right) obtained using different values of k.

2) MLBuf Training Details: The training dataset is collected from OpenROAD using the data collection process described in Section V. We use 70% of the collected data from ibex and jpeg, along with all the data from ariane, to train the model. 20% of the ibex and jpeg data is used for validation, and 10% for testing. Since all data from ariane is used in training, we exclude ariane from performance evaluation in the OpenROAD flow. To show the generalization of MLBuf, two designs, BlackParrot and MegaBoom, are held out entirely during training and used for unseen-design evaluation.

We train MLBuf using the Adam optimizer with an initial learning rate of 10^{-4}, and weight decay of 10^{-5}. The learning

rate is reduced by a factor of 0.5 every 10 epochs. The model is trained for a maximum of 1000 epochs, with early stopping applied to prevent overfitting if validation performance does not improve for 60 consecutive epochs. GELU [15] is used as the default activation function unless otherwise specified. Training is conducted on a Linux server with an NVIDIA A100 GPU. Each training step takes about one second. On average, the model training takes approximately 25 hours. The final model contains 0.46 million parameters. Details of hyperparameters are shown in Table VII.

TABLE VII: Hyperparameters of MLBuf.

Hyperparameter	**Dimension**
Input Feature Dimension (d^M)	12
Shared Feature Dimension (d^a)	12
Location Feature Dimension (d^l)	3
Electrical Feature Dimension (d^t)	5
Clustering Output Dimension (d^u)	128
Number of Clusters (d^k)	20
Buffer Type Output Dimension	6 (one-hot)
Buffer Location Output Dimension	2 (x, y)

3) Performance Comparison of Buffer-embedded Trees: We compare the performance of buffer-embedded trees predicted by *MLBuf* and those calculated by OR rsz. Specifically, we insert the predicted buffers back into the netlist based on the predicted buffer-embedded tree topology in each timing optimization iteration (13 times). The comparison of buffer areas, buffer counts, remaining slew violations and capacitance violations in each iteration is shown in Table VIII. We can see that *MLBuf* achieves ERC violation resolution that is comparable to OR rsz.

TABLE VIII: Comparison of buffer-embedded tree performance across iterations.

Design	Method	Per-Iteration Results (×13)			
		Buffer area	Buffer count	#slew violations	#cap violations
ibex	No_buf	-	-	77	174
	OR rsz	[549, 559, 608, 596, 626, 639, 637, 711, 725, 705, 700, 716, 691]	[118, 121, 126, 123, 129, 136, 139, 144, 148, 147, 145, 151, 147]	[52, 49, 46, 49, 46, 46, 47, 47, 51, 50, 47, 47, 48]	[146, 148, 146, 147, 142, 146, 144, 145, 141, 144, 145, 147, 144]
	MLBuf	[636, 616, 595, 633, 587, 648, 673, 673, 819, 787, 719, 736, 757]	[69, 73, 67, 86, 75, 76, 78, 83, 95, 93, 86, 90, 92]	[54, 54, 56, 55, 57, 59, 60, 58, 58, 61, 61, 58, 57]	[151, 148, 151, 151, 157, 156, 156, 156, 155, 156, 154, 156, 157]
jpeg	No_buf	-	-	4643	22
	OR rsz	[972, 1043, 1146, 1159, 1166, 1177, 1196, 1255, 1288, 1365, 1269, 1301, 1327]	[115, 121, 133, 136, 138, 142, 143, 148, 151, 156, 152, 153, 155]	[6, 6, 5, 5, 5, 6, 6, 5, 4, 4, 5, 5, 5]	[12, 13, 13, 16, 13, 11, 14, 12, 14, 13, 14, 13, 12]
	MLBuf	[241, 554, 602, 348, 248, 464, 241, 501, 248, 457, 431, 399, 275]	[20, 46, 59, 37, 22, 42, 21, 60, 22, 42, 40, 40, 28]	[9, 12, 14, 9, 11, 11, 10, 12, 12, 22, 22, 12, 13]	[16, 18, 20, 18, 19, 19, 18, 19, 19, 19, 20, 20, 20]

4) Ad-Hoc Baseline: This is a rule-based analytical approximation used as a baseline for comparison, as discussed in Section V-B. The approach involves (i) estimating the net wirelength using a wireload model (WLM) [6]; (ii) applying a pessimism margin of 25% to the estimated wirelength to account for placement dynamics and congestion effects; (iii) estimating the buffer count by dividing the total load capacitance (wire + sinks) in one net by the max capacitance limit of BUF_X2; (iv) adding an additional overhead factor of 20% for routing detours and inefficiencies; and (v) allocating the estimated buffer area "non-uniformly" within the bounding box of the net. All hyperparameters (25%, BUF_X2, 20%) are tuned to enhance the accuracy of estimated buffer counts by benchmarking against OR rsz, thereby improving the competitiveness of this baseline.

APPENDIX B
ARTIFACT APPENDIX

A. Abstract

This appendix describes details for using *MLBuf* artifact. It has passed MLCAD 2025 artifact evaluation and been verified for reproducibility. All data, codes and scripts are available at [38].

B. Artifact check-list (meta-information)

- **Algorithm:** Recursive learning, generative model, analytical global placement.
- **Program:** Python (PyTorch Geometric), OpenROAD, OpenSTA.
- **Compilation:** Build OpenROAD using the version provided in [38]; Python compiler.
- **Model:** *MLBuf*, a recursive learning-based generative buffering model.
- **Data set:** Training dataset collected from OpenROAD rsz, publicly available at [38].
- **Run-time environment:** Python 3 with all required libraries specified in the requirements; OpenROAD (directory OR_branch_integration/OpenROAD in [38]).
- **Hardware:** AMD EPYC 7742 64-Core Processor CPU (128 threads); NVIDIA A100-SXM4-80GB GPU.
- **Execution:** Virtual buffering during global placement.
- **Metrics:** Post-route wirelength, WNS, TNS, and power (with respect to TCP).
- **Output:** Buffer-porosity-aware global placement results.
- **Experiments:** Training the *MLBuf* model; integrating *MLBuf* into OpenROAD for *MLBuf-RePlAce*.
- **Proprietary EDA tools:** Cadence Genus 21.1, Cadence Innovus 21.1.
- **How much disk space required (approximately)?:** 30GB.
- **How much time is needed to prepare workflow (approximately)?:** 1 day.
- **How much time is needed to complete experiments (approximately)?:** 2-3 days.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** BSD3 open-source license.
- **Data licenses (if publicly available)?:** BSD3 open-source license.
- **Workflow framework used?:** OpenROAD-flow-scripts.

C. Description

1) How to access: The repository, including datasets, source codes and scripts, is available at [38]. Detailed setup and usage instructions are provided in the README file.

2) Hardware dependencies: The experiments were conducted on a Linux server with an AMD EPYC 7742 64-Core Processor CPU (128 threads), 503GB RAM and an NVIDIA A100-SXM4-80GB GPU.

3) Software dependencies: Python 3 with all packages specified in the requirements file. OpenROAD built from the version in [38].

4) Commercial software dependencies: Cadence Genus 21.1 for synthesis; Cadence Innovus 21.1 for commercial-flow validation.

5) Data sets: The training datasets were collected from OpenROAD rsz and are publicly available at [38].

6) Models: MLBuf, a recursive learning-based generative buffering model.

D. Installation

Detailed installation instructions are provided in the README file of the GitHub repository [38]. These include Python environment setup, dependency installation, dataset, and execution scripts.

E. Experiment workflow

The complete workflow is described in the paper and the README file in the GitHub repository [38]. It includes training *MLBuf*, building OpenROAD from the specified version, and conducting *MLBuf-RePlAce* to produce buffer-porosity-aware global placement results.

F. Evaluation and expected results

Evaluation results are provided in the paper, including performance of predicted buffer-embedded trees and the post-route PPA results of *MLBuf-RePlAce*.

G. Experiment customization

Users can customize experiments by adjusting *MLBuf* hyperparameters such as clustering output dimension and the number of clusters. Additionally, users can adjust the overflow_list in the provided scripts, which controls the number of virtual buffering (i.e., ERC fixing in Figure 1 of the paper) iterations performed during global placement.

H. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- http://cTuning.org/ae/submission-20201122.html
- https://github.com/ml-eda/artifact-evaluation/