BufFormer: A Generative ML Framework for Scalable Buffering

Rongjian Liang
rliang@nvidia.com
NVIDIA, US

Siddhartha Nath
siddhartha.nath@gmail.com
NVIDIA, US

Anand Rajaram
arajaram@nvidia.com
NVIDIA, US

Jiang Hu
jianghu@tamu.edu
Department of ECE & CSE
Texas A&M University, US

Haoxing Ren
haoxingr@nvidia.com
NVIDIA, US

ABSTRACT

Buffering is a prevalent interconnect optimization technique to help timing closure and is often performed after placement. A common buffering approach is to construct a Steiner tree and then buffers are inserted on the tree based on Ginneken-Lillis style algorithm. Such an approach is difficult to scale with large nets. Our work attempts to solve this problem with a generative machine-learning (ML) approach without Steiner tree construction. Our approach can extract and reuse knowledge from high quality samples and therefore has significantly improved scalability. A generative ML framework, BufFormer, is proposed to construct abstract tree topology while simultaneously determining buffer sizes & locations. A baseline method, FLUTE-based Steiner tree construction followed by Ginneken-Lillis style buffer insertion, is implemented to generate training samples. After training, BufFormer can produce solutions for unseen nets highly comparable to baseline results with a correlation coefficient 0.977 in terms of buffer area and 0.934 for driver-sink delays. On average, BufFormer-generated tree achieves similar delays with slightly larger buffer area. And up to 160X speedup can be achieved for large nets when running on a GPU over the baseline on a single CPU thread.

CCS CONCEPTS

• **Hardware → Electronic design automation**; • **Physical design (EDA)**;

KEYWORDS

buffer insertion, timing optimization, interconnect optimization, generative model, machine learning

ACM Reference Format:
Rongjian Liang, Siddhartha Nath, Anand Rajaram, Jiang Hu, and Haoxing Ren. 2023. BufFormer: A Generative ML Framework for Scalable Buffering. In *28th Asia and South Pacific Design Automation Conference (ASPDAC '23), January 16–19, 2023, Tokyo, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3566097.3567900

ASPDAC '23, January 16–19, 2023, Tokyo, Japan

ACM ISBN 978-1-4503-9783-4/23/01.
https://doi.org/10.1145/3566097.3567900

1 INTRODUCTION

Buffering is a critical interconnect optimization technique for timing closure [12]. With the increasing dominance of interconnect delay over gate delay, it is not uncommon that more than 30% cells are buffers in VLSI circuits at advanced technology nodes [19].

Due to its great importance, buffering algorithms have been heavily studied, from net-level buffering [11, 21, 27] to circuit-level optimization [20, 26] as well as integration with other optimization techniques and design steps [7, 18]. Net-level buffering is the backbone of other buffering techniques, and is the target of this work. A common practice is to construct a Steiner minimum tree [2] or timing-driven tree [13] first, followed by wire segmenting [9] on the tree to generate candidate buffer locations, and then perform Ginneken-Lillis style algorithms [15] to determine final buffer placement. The joint space of tree generation and buffer insertion can be huge [3], especially for large nets. Hence, existing algorithms often use modeling approximations and heuristics to achieve acceptable runtime [2, 10, 21]. Despite this, previous algorithms tend not to scale well for large nets. It is reported in [2], which proposes a fast Steiner tree generation package FLUTE, that its runtime complexity is $O(n \log(n))$ (n is the number of sinks) and it takes 0.1s-13s to construct Steiner trees with different qualities for a net with 100 sinks. It is proved in [24] that the timing-driven minimum cost buffer insertion problem is NP-complete. Lillis et al.[16] proposes a dynamic programming-based solution that runs in pseudo-polynomial time and a fully polynomial approximation method is developed in [21]. It would be even more runtime-expensive to navigate the joint solution space of tree generation and buffer insertion. It is not uncommon to take an hour to conduct buffering for an industrial circuit with over 1 million nets.

To address the scalability challenge for large nets, we propose a generative machine-learning (ML) approach [25]. Generative models generate new data instances by capturing the distribution of data. With the help of the powerful modeling capability of ML, generative ML approach can extract and reuse knowledge from high quality samples, and is potentially more scalable due to its advanced computational paradigms (e.g., ML models with GPU acceleration) [17]. We deploy generative ML for constructing an abstract tree topology while simultaneously determining the locations & sizes of buffers, without Steiner tree construction. We name such a tree *buffer-embedded tree*. A recent work develops a generative ML method for gate sizing and achieves over 1000X speedup [22]. While gate sizing on a timing path can be treated as a sequence generation problem that has been well-studied by the

ML community, the following new challenges need to be solved to enable an efficient ML solution to buffer-embedded tree generation: (1) optimization in the complex joint space of tree topology construction, buffer type selection and buffer location determination; (2) consideration of complicated conditional information of the driver and sinks, including their physical, electrical, timing and parity attributes; (3) taking advantage of archived buffer tree samples from legacy designs to save the runtime and license cost of evoking design tools/flows.

We present a novel generative ML framework for buffering, named BufFormer, to address the aforementioned challenges. BufFormer enables efficient buffer-embedded tree generation with a layer-by-layer clustering-based process. The information of the driver and all sinks is captured by a transformer[8]-based network. And a self-supervised [25] training scheme is developed to allow learning from archived data. Our main contributions are summarized as follows:

(1) To our best knowledge, this work is the first successful attempt at generative ML-based buffering, a critical problem in EDA. Our approach learns how to conduct buffering by capturing the distribution of high quality samples.
(2) A generative ML framework, BufFormer, is proposed for buffering. It incorporates physical, electrical, timing and parity information of cells to generate buffer-embedded trees without Steiner tree construction.
(3) To generate training samples, a baseline method using the FLUTE package for Steiner tree construction and a Ginneken-Lillis style algorithm [16] for buffer insertion is implemented. After training, BufFormer can produce solutions for unseen nets very close to baseline results with a correlation coefficient 0.977 in terms of buffer area and 0.934 for driver-sink delays. On average, BufFormer-generated tree achieves almost the same delays with slightly larger buffer area. It can generate a buffer-embedded tree for a net with 120 sinks within 0.1 s when running on a GPU and achieve up to 160X speedup over the baseline on a single CPU thread. BufFormer might serve as a speedy buffering engine in early design iterations to facilitate fast timing closure.

2 PRELIMINARY

2.1 Transformer and Self-Attention

Transformer is a generative neural network architecture relying entirely on attention mechanism for sequence-to-sequence mapping. Generally speaking, attention mechanism permits neural network models to focus on the most relevant parts of data while ignoring other parts in a trainable manner. Further, self-attention on the input sequence, i.e., calculating attention of all other inputs w.r.t. each input, allows inputs interact with each others to learn representations capturing both global and local information. Mathematically, the attention operation for input matrices $(\mathbf{Q}, \mathbf{K}, \mathbf{V})$ is calculated as:

$$Attention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = Softmax(Mask(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}))\mathbf{V}, \quad (1)$$

where $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ are the concatenation of query, key, and value vectors, which are obtained by applying linear transformations to input vectors. And d is the dimension of the query/key/value vectors. Here, *Mask* refers to adding negative infinity to specific elements such that the corresponding elements will become zeros after the $Softmax$ operation, meaning no effect on the output. *Mask* operators can be customized by users to integrate prior knowledge, e.g., specific part of inputs is not relevant to another part of inputs, to the models.

2.2 Self-Supervised Learning

Roughly speaking, self-supervised learning predicts part of the data from any observed parts [25]. In our problem, the driver and sinks are the observed parts of a buffer-embedded tree and self-supervised learning is utilized to predict its remaining parts, as shown in Figure 1. Unlike conventional supervised learning or reinforcement learning methods requiring some sorts of labels or rewards (e.g., timing performance) from the environment, self-supervised approach can generate "label" from tree samples themselves. To be specific, the archived tree samples contains the information of what a buffer-embedded tree should look like given the driver and sinks as well as delay target. Self-supervised learning utilizes such information as "labels" to train models such that they can generate new trees with the same distribution to the archived samples. In this way, we can take advantage of samples from legacy designs to save the runtime and license cost of evoking design tools/flows.

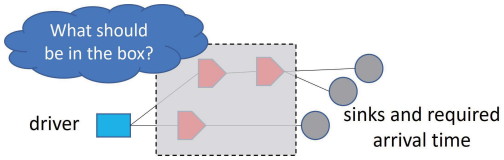

Figure 1: Illustration of self-supervised learning.

2.3 Problem Formulation

The net-level buffering problem can be defined as follows. Given

- a driver cell, its location and input slew,
- a set of sinks, and for each sink its parity, location, and the delay target from the input pin of driver and the sink (this is actually the required arrival time for each sink assuming the arrival time at the input pin of driver is 0),
- a library of buffers and inverters,
- the timing information and input capacitance for each cell, including the buffer/inverter cells,
- electrical information about wiring for estimating resistance and capacitance,

the goal is to determine a buffer-embedded tree such that the total cost (measured by the total buffer/inverter areas) is minimized while the delay targets, parity constraint (i.e. the number of inverters on the path from the driver to the sink is even if and only if the sink has parity +), capacitance and slew limits are satisfied.

In conventional methods, Steiner or timing-driven trees are usually used to assist the buffering process and the routing information on the tress is dropped after buffer insertion, especially in the timing optimization steps at pre-routing stages. In contrast, trees are not required as input to our approach and our method does not generate the actual routing for interconnects.

In our experiment setting, cell timing is estimated via look-up table and the Elmore delay analysis model is utilized for estimating wire delay. It is noteworthy that, technically, our method is compatible with any timing models as long as a large set of high quality samples optimized under the timing model can be collected. In addition, for evaluation purpose, Steiner trees are constructed for each interconnect in our generated buffer-embedding trees to estimate wire delays. Please note that the Steiner tree construction is not required when applying our method.

3 METHODOLOGY

3.1 Overview of BufFormer Framework

BufFormer is a generative ML framework for buffer-embedded tree generation. It consists of a clustering-based tree generation process, a transformer-based BufFormer-Net and a self-supervised training scheme, as shown in Figure 2. BuFFormer-Net is trained with archived samples in a self-supervised learning manner. Trained BufFormer-Net guides the buffer-embedded trees generation process. Technical details are elaborated in the following.

3.2 Tree Generation Process

Our tree generation process is shown in Figure 2, which has three important properties as follows.

(1) Recursive process. The tree consisting of only the driver and sinks is viewed as the original buffer-embedded tree. Buffers are inserted to the tree recursively to improve performance. In this way, the constraint that the tree must have the driver cell as root and sinks as leaves can be naturally satisfied.
(2) Layer-by-layer bottom-up process. Similar to classical Ginneken-Lillis style algorithms, our tree generation process inserts buffers in a bottom-up manner. We define the height of a cell in a buffer-embedded tree as the number of edges on the longest path from itself to a sink, and a layer of buffers as a set of buffers with the same height. Unlike Ginneken-Lillis algorithm processing one candidate buffer location at a time, our generation process inserts a layer of buffers in their proper locations simultaneously without the Steiner tree generation and wire segmenting, as depicted in fig. 2 (a). In such a way, a buffer tree of height N can be constructed in N steps.
(3) Clustering-based process. The buffer-embedded tree is constructed in a hierarchical clustering manner. Given the driver and sinks with delay targets, our BufFormer-Net aggregates information on the driver and all sinks to determine what to cluster and whether to insert buffer for each cluster. Therefore the decision for one cluster correlates to the decisions for other clusters. After the first layer clustering, the delay targets of newly created buffers will be updated and the new buffers will be regarded as dummy sinks that serve as input to the next layer of buffering. The reason is that the inserted buffers will shield the effects of downstream cells/wires and we only need to update the delay targets by considering the delays of downstream cells/wires. This property is also utilized by Ginneken-Lillis style algorithms to simplify the problem. Note that if no buffer is required for a cluster, then sinks in the cluster are untouched and will go through the next layer of clustering. The stop condition for our tree generation process is that no more buffer is required for any cluster. It is noteworthy that one sink can also form a cluster, which enables BufFormer to handle the problem of repeater insertion along a long wire.

To sum up, our generation process can generate buffered-embedded tree with the driver as root and sinks as leaves, getting rid of the construction of Steiner tree and wire segmenting. Further, a tree of height N (i.e., the height of the driver is N) can be constructed in N steps, which is more scalable for large nets than classical Ginneken-Lillis style algorithms.

3.3 BufFormer-Net

3.3.1 Transformer-Based Architecture. During each layer of buffer insertion in our generation process, BufFormer-Net is responsible for four tasks, i.e., given the information of the driver and sinks, determining 1) the clustering of sinks, 2) the sizes of buffers/inverters (or *None* to indicate that no buffer is needed) for each cluster, 3) the buffer locations, and 4) the delay targets for the newly inserted buffers. All these tasks can be viewed as sequence-to-sequence "translation" between different domains. Also, the clustering task requires to aggregate the information of the driver and all sinks. Thus, we customize a transformer-based architecture for BufFormer-Net.

As shown in Figure 2 (b), BufFormer-Net has the Self-Attention0 module, which consists of a series of self-attention layers, learn representations of sinks by incorporating the information of the driver and sinks. The learnt representations are shared and used for all tasks. Then there are individual modules for each task, e.g., FCL0 module (Here, FCL is the short for Fully Connected Layer) for the clustering task. We let these four tasks share the same sink representations mainly for two reasons. First, we argue that these tasks are related to each other. Hence, the learnt representations for one task might benefit other tasks too. Also, sharing the Self-Attention0 module can help reduce the total model parameters, consequently less memory usage and faster training and inference.

BufFormer clusters sinks first, whose results are used to guide other tasks. To be specific, for each sink cluster, the prediction of buffer size/location/delay target might only need to involve the sink representations in the corresponding cluster. Such priors are integrated into BufFormer-Net by constructing attention masks for Self-Attention1,2,3 modules according to clustering results, as depicted in Figure 2 (b). Further, the buffer size prediction results are used for buffer location and delay target prediction, while results of location prediction are also utilized for delay target prediction.

Note that BufFormer-Net feeds learnt sink representations to a clustering algorithm to generate final clusters, as depicted in Figure 2 (b). A simple clustering algorithm, named Connection Clustering, is developed. Firstly, the cosine similarity between representations of every two sinks are calculated. Sink pairs having similarity higher than a given threshold are regarded connected. Then, connected components among sinks are detected. If a connected component contains sinks with different parities, then split the component into two parts according to parities of sinks. Finally, each component is viewed as one cluster. Such a clustering algorithm does not need pre-defined number of clusters and can automatically enforce the fulfillment of the parity constraint. We implement Connection Clustering as a series of matrix operations, which can be conveniently accelerated by GPU.

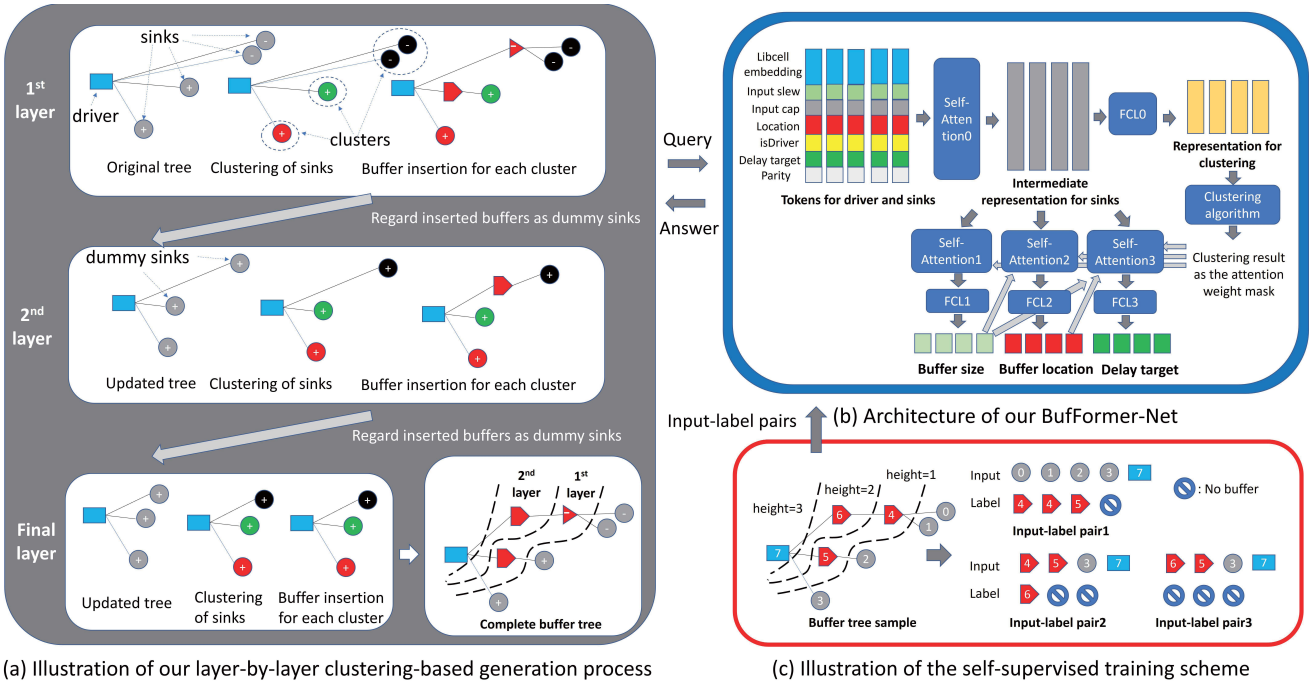

Figure 2: Illustration of our BufFormer framework, which consists of a layer-by-layer tree generation process, a transformer-based BufFormer-Net, and a self-supervised training scheme. (a) shows a 3-layer generation process for a tree of height 3. (b) shows how BufFormer-Net make decisions for the 1^{st} layer buffer insertion by integrating the information of the driver and 4 sinks. (c) illustrates that 3 input-label pairs can be constructed from a tree sample of height 3.

With respect to capacitance and slew limits, they are implicitly modelled by training BufFormer-Net with legal samples.

3.3.2 Feature Engineering. We incorporate physical, electrical, timing and parity information into input features to BufFormer-Net. More specifically, the input to BufFormer-Net is a sequence of feature vectors, each of which corresponds to the driver or one (dummy) sink, as depicted in Figure 2 (b). Each feature vector consists of a trainable embedding for the cell type, input slew (-1 for sinks), input capacitance (-1 for the driver), relative location and distance to the driver, a flag indicating whether it is a driver or a sink, the delay target (0 for the driver), and parity (+ for the driver). Note that input normalization is applied since the magnitudes of different features vary greatly.

3.4 Self-Supervised Training Scheme

3.4.1 Training Data Preparation. Essentially, self-supervised training scheme constructs labels from tree samples themselves and then train BufFormer-Net in a supervised-learning manner. Given a buffer-embedded tree, the heights of cells in the tree can be found using a depth-first search algorithm. For a tree of height N, N input-label pairs can be constructed, as illustrated in Figure 2(c). Specifically, input in the first input-label pair consists of the set of sinks as well as the driver, while labels are the parent buffers or *None* denoting no buffer. For the i-th ($1 < i \leq N$) pair, the input consists of the set of (dumpy) sinks with height $i-1$, the driver, and the cells in the input set of the $(i-1)$-th pair that have *None* as label.

3.4.2 Training Losses. A contrastive loss is utilized for the clustering task to push sinks belonging to the same cluster closer in the representation space while simultaneously pushing apart sinks from different clusters. Firstly, the cosine similarity between a pair of sink representations $(\boldsymbol{y}_i, \boldsymbol{y}_j)$ output by BufFormer-Net is calculated as follow:

$$s(\boldsymbol{y}_i, \boldsymbol{y}_j) = 0.5 \times \left(\frac{\boldsymbol{y}_i \boldsymbol{y}_j}{|\boldsymbol{y}_i||\boldsymbol{y}_j|} + 1\right). \tag{2}$$

Note that $s(\boldsymbol{y}_i, \boldsymbol{y}_j) \in [0, 1]$ for any $\boldsymbol{y}_i, \boldsymbol{y}_j$. And it can be interpreted as the probability of sinks i, j belonging to the same cluster. For notational convenience, we define $p(\boldsymbol{y}_i, \boldsymbol{y}_j)$ as:

$$p(\boldsymbol{y}_i, \boldsymbol{y}_j) = \begin{cases} s(\boldsymbol{y}_i, \boldsymbol{y}_j) & \text{if sinks } i, j \text{ belong to the same cluster,} \\ 1 - s(\boldsymbol{y}_i, \boldsymbol{y}_j) & \text{Otherwise.} \end{cases} \tag{3}$$

The clustering loss is defined as

$$\mathrm{L}(\boldsymbol{y}_i, \boldsymbol{y}_j) = -\log[p(\boldsymbol{y}_i, \boldsymbol{y}_j)]. \tag{4}$$

Such a loss function will push $s(\boldsymbol{y}_i, \boldsymbol{y}_j)$ to 1 if nodes i, j belong to the same cluster and to 0 if not.

The buffer size prediction task is a $(M+1)$-class classification problem, where M is the total number of buffers/inverters in the library, and an extra class for *None*. We observe high imbalance between the usage frequencies among different buffer/inverter library cells in our data set, therefore we deploy Focal loss [23], which is an enhancement to the classical cross entropy loss to handle class imbalance, for the buffer size prediction task. The raw classification output for a buffer is $z \in \{[z_0, z_1, \cdots z_M] \mid 0 \leq z_0, z_1, \cdots, z_M \leq 1 \text{ and } z_0 + z_1 + \cdots, z_M = 1\}$, which describes the probabilities of

this buffer belonging to each of the $M+1$ classes. Denote the labels as $l \in \{0, 1, \cdots M\}$, then focal loss can be calculated as follow:

$$\mathrm{FL}(z, l) = \sum_{i=0}^{M} -(1 - p_z^i)^\gamma log(p_z^i), \tag{5}$$

$$p_z^i = \begin{cases} z_i, & if \quad i = l \\ 1 - z_i, & otherwise. \end{cases} \tag{6}$$

$(1 - p_z^i)^\gamma$ is a modulating factor for down-weighting well-classified samples. $(1-p_z^i)$ near 0 indicates easy-to-classify samples while $(1-p_z^i)$ near 1 means difficult-to-classify. When a sample is misclassified and p_z^i is small, the modulating factor is near 1 and the loss is unaffected. As $p_z^i \to 1$, the factor goes to 0 and the loss for well-classified samples is down-weighted. The parameter γ adjusts the rate at which easy samples are down-weighted,.

For the buffer location and the delay target prediction tasks, we use the mean-square-error loss. Note that the predicted delay target of each buffer will be used in the features for the prediction of next level clustering during inference.

3.4.3 Multi-Objective Training. The training of BufFormer-Net is a multi-objective optimization problem since four losses, each of which corresponds to one task, are to optimize. A common practice is to optimize a linear combination of per-task losses. However, it is not easy to find the proper weights for the losses to balance the priorities of different tasks. A gradient-based neural network training technique that can optimize a collection of objectives is proposed in [6]. For the individual modules for each task, e.g., FCL0 module for clustering, their parameters are updated according to the gradients of the corresponding loss. As for the shared network parameters, i.e., parameters in Self-Attention0, they are updated by following the ideas in [6] as follows. Firstly, the gradients of each loss w.r.t. the parameters in Self-Attention0 are computed. Then a minimum-norm vector in the convex hull of the set of gradient vectors for our four tasks is found. Finally, parameters are updated in the direction of the minimum-norm vector. It has been shown in [6] that such gradient update approach essentially uses adaptive weights and optimizes an upper bound for the multi-objective loss.

4 EXPERIMENTAL VALIDATION

4.1 Experiment Setups

Experiments are conducted in the OpenPhySyn platform [1]. Five types of buffers and four types of inverters from NanGate45 open cell library are used for buffering. Input capacitance of these cells varies from 0.78 fF to 49.2 fF, and area varies from 0.8 μm^2 to 13.0 μm^2. The maximum load capacitance is set to 100 fF and we use the slew limits defined by the cell library. To generate training samples and benchmark our performance, a baseline method using the FLUTE package for Steiner tree generation and a Ginneken-Lillis style algorithm [16] for buffer insertion is implemented. Given a Steiner tree and candidate buffer locations on it, the Ginneken-Lillis style algorithm can find a set of buffering solutions that achieve different trade-offs between delay and buffer area. Considering the trade-off between solution quality and runtime, candidate buffer locations are inserted every 60μm (which is around 20X of the width of a buffer) along each edge of the Steiner tree, all Steiner points and all turning points.

Table 1: Characteristics of Training and Testing Samples

characteristic	train set	test set
net count	23083	1620
tree count	343004	32031
sink count range	[1,150]	[0,98]
buffer area range (μm^2)	[0,151]	[0,99]
driver-sink delay range (ps)	[0,1128]	[0,609]
HPWL range (μm)	[0, 2214]	[0, 2796]

To collect a large training set, we sample artificial net instances and then invoke the FLUTE+Lillis baseline method to generate buffer tree samples. Note that a set of buffer trees achieving different trade-offs between buffer area and delay is collected for each net. For simplicity but without loss of generality, we also use the library of buffers/inverters for the drivers and sinks. The characteristics of collected samples are shown in Table 1. The training set contains over 300K buffer trees while the testing set contains around 30K samples. The achieved driver-sink delays by the baseline method are recorded as the delay targets for the training and evaluation of our method. This ensures that the delay targets are achievable. We tend to collect buffer trees for large nets over smaller ones as training samples, because the buffer trees for large nets are more informative as their sub-trees can be viewed as solutions for small nets.

Our BufFormer framework is built with the PyTroch package. In the default version of BufFormer-Net, 6 layers of 8-head self-attention are used for Self-Attention0, 3 layers of 8-head-attention for each of Self-Attention1,2,3, and 3 fully connected layers for each of FCL0,1,2,3. The dimension of intermediate representations is set to 256. And the batch size for training and testing are 128 and 256, respectively. The FLUTE+Lillis baseline method runs on a single Intel Xeon CPU thread, while BufFormer-Net runs on a NVIDIA Tesla V100 GPU.

Both layer-wise performance metrics and entire tree performance metrics are evaluated. There are four tasks for each layer of buffer insertion, i.e., clustering of sinks, prediction of buffer library cell, of buffer location, and of delay target. For measuring layer-wise performance, we utilize pair-wise accuracy for clustering performance, classification accuracy for library cell prediction, and root mean square error (RMSE) for buffer location and delay target prediction. Pair-wise clustering accuracy is obtained by dividing the number of sink pairs that BufFormer-Net correctly predicts whether they are in the same cluster by the total number of sink pairs. Classification accuracy refers to the number of correctly predicted buffers divided by the total number of buffers.

To evaluate the quality of an entire buffer-embedded tree generated by BufFormer, we measure the total buffer area and the achieved driver-sink delays for all driver-sink pairs, and compare them with the area and delays achieved by the baseline method. Their correlation coefficient, and the mean and standard deviation (std.) of the difference between ML generated trees and baseline trees are calculated. Note that the difference is computed as:

$$area/delay\ achieved\ by\ BufFormer\ -\ baseline\ area/delay.$$

Negative values mean better performance than the baseline results.

Table 2: Results of Ablation Studies

factors	method	cluster acc	libcell acc	loc RMSE (μm)	delay tar. RMSE (ps)	buf. area diff. (μm^2)			driver-sink delay diff.(ps)		
						cor	mean	std	cor	mean	std
	default	**92.6%**	**91.6%**	**34**	**10**	**0.977**	**4.8**	**4.9**	**0.934**	**0**	**25**
data amount	40%	91.1%	88.7%	36	11	0.964	4.6	5.9	0.762	7	51
	25%	87.0%	84.5%	40	14	0.939	1.3	7.1	0.761	8	50
model size	larger	93.1%	92.5%	34	10	0.975	6.3	5.3	0.931	0	25
	smaller	92.9%	90.7%	34	10	0.970	4.7	5.4	0.927	0	26
train loss	weighted	82.9%	95.8%	34	8	0.976	4.1	4.5	0.873	0	35
model arch.	separate	90.8%	95.5%	36	8	0.964	6.8	6.1	0.762	7	51
clustering algorithm	AC	/	/	/	/	0.974	4.2	4.9	0.916	0	28
	AP	/	/	/	/	0.972	6.1	5.4	0.905	0	30
	DBSCAN	/	/	/	/	0.932	-2.1	7.9	0.676	11	67

4.2 Experiment Results

4.2.1 Comparison with Baseline Method. As shown in Figure 3 and Table 2, BufFormer can generate buffer-embedded trees for unseen nets with highly comparable quality to the baseline results. Specifically, as for layer-wise performance, it achieves 92.6% clustering accuracy, 91.6% library cell prediction accuracy, 34 μm RMSE in location prediction and 10 ps RMSE in delay target prediction. For the performance of entire buffer trees, the correlation coefficient between ML-generated trees and the one generated by the baseline is as high as 0.977 in terms of buffer area and 0.934 for driver-sink delays. Compared with baseline results, ML-generated tree achieves almost the same driver-sink delays with slightly larger buffer area. The average buffer area overhead is 4.8 μm^2, around the area of a middle size buffer. As shown in Figure 3, the divergence of buffer area becomes higher for trees with buffer area larger than 60 μm^2. One reason might be that we do not have enough training samples with large buffer area. Such divergence could be mitigated and the buffer area overhead would decrease if BufFormer is trained with a greater number of large buffer tree samples.

It is interesting to note that, though on average ML-generated tree achieves the similar driver-sink delays with slightly larger buffer area, there are cases where ML-generated trees outperform the baseline results in terms of both buffer area and delay. Figure 5 depicts one of the examples. In future work we will systematically investigate the impact of refining the training set recursively by replacing the original buffer trees by the ML-generated trees that are of greater quality.

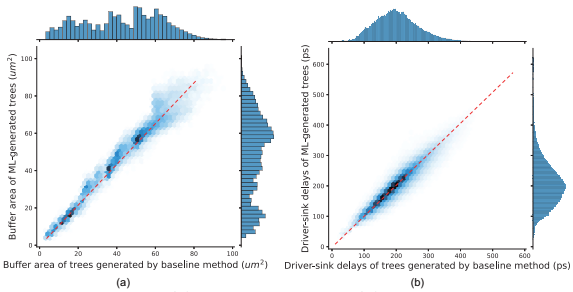

Figure 3: Comparing (a) buffer areas and (b) all achieved driver-sink delays of ML-generated buffer trees with baseline trees.

As discussed in Section 3, our tree generation process guarantees the satisfaction of the polarity constraint. We empirically observe that ML-generated buffer trees also satisfy the slew and capacitance limits everywhere. It implies that generative ML model can capture such constraints by learning the distribution of legal solutions.

Table 3: Speedup of Our Method over Baseline for Different Net Sizes

group idx	1	2	3	4	5
sink count	[1,3]	[31,33]	[61,63]	[91,93]	[121,123]
avg. HPWL (μm)	40	225	348	470	590
avg. speedup	0.2	38	53	125	165

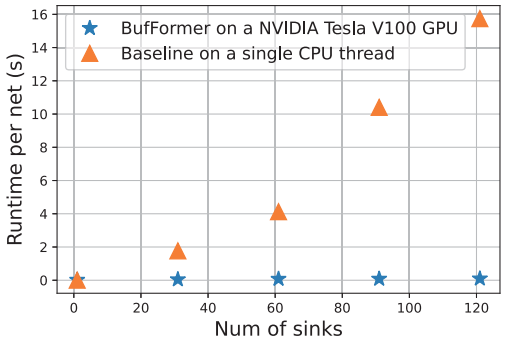

Figure 4: Runtime comparison between BufFormer and the baseline.

To study the scalability of our method, we apply BufFormer and the baseline to five groups of nets with different sizes. Table 3 shows the characteristics of nets and the speedup of our method over the baseline. The absolute runtime per net is depicted in Figure 4. The baseline runs on a single CPU thread and the runtime for Steiner tree generation and buffer insertion are measured. For BufFormer, we assume that the features of the driver and sinks are already extracted and stored in a matrix form, and we measure the runtime taken by BufFormer to generate an entire buffer-embedded tree. Note that the input features for BufFormer can be easily computed. As for the FLUTE+Lillis baseline, we observe that the runtime grows fast as the size of nets increases. And the majority runtime is spent on the timing-driven minimum cost buffer insertion. As for BufFormer, it can generate buffer tree for a net with 121 sinks

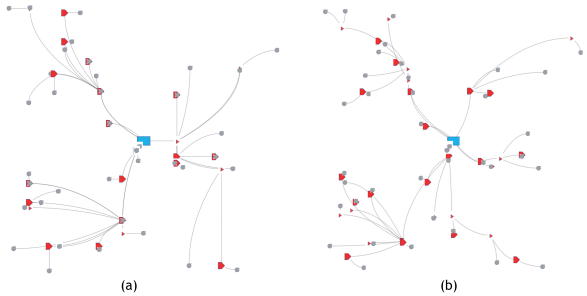

Figure 5: Snapshot of (a) a buffer-embedded tree generated by the baseline method and (b) a tree by BufFormer for the same net. Blue box is the driver, gray circles are sinks, and red shapes represent buffers/inverters. The total buffer area and mean driver-sink delay for the baseline tree are $31.7\,\mu m^2$ **and** 155 ps**, respectively, while the tree generated by BufFormer reduces the buffer area by** $3.2\,\mu m^2$ **and decreases the mean delay by** 8 ps**.**

within 0.1s, with a speedup up to 160X over the baseline. Our speedup is obviously greater than the fast buffering method [21], which achieves 4.6X speedup over Ginneken-Lillis style algorithms and requires a Steiner tree as input. The scalability advantage of BufFormer mainly comes from our efficient tree generation process as well as GPU acceleration. BufFormer can generate a tree of height N in N steps, which is fairly scalable with large nets. Also, its neural network backbone enables GPU acceleration for significant speedup that is difficult to be achieved by multi-threading programming used in traditional methods, since GPUs can bring massive parallelism, i.e., great throughput.

4.2.2 Ablation Studies. The following ablation studies are conducted to identify important factors for BufFormer performance.

(1) Data amount. Besides using the full training set, we train BufFormer with 40% data and 25% data.
(2) Model size. A larger BufFormer-Net (dimension of intermediate representations = 516) and a smaller one (dimension = 128) are implemented.
(3) Train loss. A linear combination of per-task losses is used for training rather than our multi-objective training method.
(4) Model architecture. Instead of using a shared model, individual models are developed for each of four tasks: clustering, buffer library cell, buffer location and delay target prediction.
(5) Clustering algorithms. Besides our default Connection Clustering algorithm, we also integrate the Agglomerative Clustering [5], Affinity Propagation [4] and DBCAN [14] algorithms into our BufFormer framework. All of them can automatically determine the number of clusters.

Table 2 shows the ablation study results. It seems that the most influential factor is the data amount. Greater amount of training data help boost the performance of BufFormer. The default model size seems to be a proper choice given current amount of training data, since increasing the model size does not improve performance. Using a shared model and deploying our multi-objective training scheme contribute to the good performance of BufFormer. As for clustering algorithms, while DBCAN delivers much worse results, Agglomerative Clustering and Affinity Propagation produce slightly worse results to the default Connection Clustering algorithm.

5 CONCLUSION AND FUTURE DIRECTIONS

This work is an attempt to solve buffering problem with a generative ML framework, named BufFormer. BufFormer generates buffer-embedded trees without Steiner tree construction by learning from high quality samples. After training with tree samples generated by a FLUTE+Lillis baseline algorithm, it can produce solutions for unseen nets highly comparable to baseline results with a correlation coefficient 0.977 in terms of buffer area and 0.934 for driver-sink delays. BufFormer-generated tree achieves similar delays with slightly larger buffer area. And up to 160X speedup can be achieved for large nets when running on a GPU over the baseline on a single CPU thread. BufFormer might serve as a speedy buffering engine in early design iterations. We plan to extend the BufFormer framework to handle realistic layout environment constraints (e.g., placement and routing congestion) and circuit-level optimization for industrial designs.

REFERENCES

[1] A. Agiza and S. Reda. 2020. Openphysyn: An open-source physical synthesis optimization toolkit. In *Proc. WOSET*.
[2] C. Chu and Y-C. Wong. 2007. FLUTE: Fast lookup table based rectilinear Steiner minimal tree algorithm for VLSI design. *TCAD* 27, 1 (2007), 70–83.
[3] J. Cong and X. Yuan. 2000. Routing tree construction under fixed buffer locations. In *Proc. DAC*. 379–384.
[4] D. Dueck. 2009. *Affinity propagation: clustering data by passing messages*. Citeseer.
[5] F. Murtagh and P. Legendre. 2014. Ward's hierarchical agglomerative clustering method: which algorithms implement Ward's criterion? *J. Classif.* 31, 3 (2014), 274–295.
[6] O. Sener and V. Koltun. 2018. Multi-task learning as multi-objective optimization. *NeurIPS* 31 (2018).
[7] A. Stefanidis *et al.* 2020. Autonomous application of netlist transformations inside lagrangian relaxation-based optimization. *TCAD* 40, 8 (2020), 1672–1686.
[8] A. Vaswani *et al.* 2017. Attention is all you need. *NeurIPS* 30 (2017).
[9] C. J. Alpert *et al.* 1997. Wire segmenting for improved buffer insertion. In *Proc. DAC*. 588–593.
[10] C. J. Alpert *et al.* 2001. Steiner tree optimization for buffers, blockages, and bays. *TCAD* 20, 4 (2001), 556–562.
[11] C. J. Alpert *et al.* 2006. Accurate estimation of global buffer delay within a floorplan. *TCAD* 25, 6 (2006), 1140–1145.
[12] C. J. Alpert *et al.* 2008. *Handbook of algorithms for physical design automation*. CRC press.
[13] C. J. Alpert *et al.* 2018. Prim-Dijkstra revisited: Achieving superior timing-driven routing trees. In *Proc. ISPD*. 10–17.
[14] E. Schubert *et al.* 2017. DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. *TODS* 42, 3 (2017), 1–21.
[15] G. Van *et al.* 1990. Buffer placement in distributed RC-tree networks for minimal Elmore delay. In *Proc. ISCAS*. 865–868.
[16] J. Lillis *et al.* 1996. Optimal wire sizing and buffer insertion for low power and a generalized delay model. *JSSC* 31, 3 (1996), 437–447.
[17] J. Yan *et al.* 2022. Towards Machine Learning for Placement and Routing in Chip Design: a Methodological Overview. *arXiv preprint arXiv:2202.13564* (2022).
[18] M. L. Mui *et al.* 2004. A global interconnect optimization scheme for nanometer scale VLSI with implications for latency, bandwidth, and power dissipation. *T-ED* 51, 2 (2004), 195–203.
[19] P. Saxena *et al.* 2004. Repeater scaling and its impact on CAD. *TCAD* 23, 4 (2004), 451–463.
[20] R. Chen *et al.* 2005. Efficient algorithms for buffer insertion in general circuits based on network flow. In *Proc. ICCAD*. 322–326.
[21] S. Hu *et al.* 2009. A fully polynomial time approximation scheme for timing driven minimum cost buffer insertion. In *Proc. DAC*. 424–429.
[22] S. Nath *et al.* 2022. Invited: Generative Self-Supervised Learning for Gate Sizing. In *Proc. DAC*.
[23] T. Y. Lin *et al.* 2017. Focal loss for dense object detection. In *Proc. ICCV*. 2980–2988.
[24] W. Shi *et al.* 2004. Complexity analysis and speedup techniques for optimal buffer insertion with minimum cost. In *Proc. ASP-DAC*. 609–614.
[25] X. Liu *et al.* 2021. Self-supervised learning: Generative or contrastive. *TKDE* 01 (2021), 1–1.
[26] Z. Jiang *et al.* 2008. Circuit-wise buffer insertion and gate sizing algorithm with scalability. In *Proc. DAC*. 708–713.
[27] Z. Li *et al.* 2005. Making fast buffer insertion even faster via approximation techniques. In *Proc. ASP-DAC*, Vol. 1. 13–18.