DAGSizer: A Directed Graph Convolutional Network Approach to Discrete Gate Sizing of VLSI Graphs

CHUNG-KUAN CHENG, CHESTER HOLTZ, ANDREW B. KAHNG, BILL LIN, and UDAY MALLAPPA, University of California, San Diego, USA

The objective of a leakage recovery step is to make use of positive slack and reduce power by performing appropriate standard-cell swaps such as threshold-voltage (V_{th}) or channel-length reassignments. The resulting engineering change order netlist needs to be timing clean. Because this recovery step is performed several times in a physical design flow and involves long runtimes and high tool-license usage, previous works have proposed graph neural network–based frameworks that restrict feature aggregation to three-hop neighborhoods and do not fully consider the directed nature of netlist graphs. As a result, the intermediate node embeddings do not capture the complete structure of the timing graph. In this article, we propose *DAGSizer*, a framework that exploits the *directed acyclic* nature of timing graphs to predict cell reassignments in the discrete *gate sizing* task. Our DAGSizer (Sizer for DAGs) framework is based on a node ordering-aware recurrent message-passing scheme for generating the latent node embeddings. The generated node embeddings absorb the complete information from the fanin cone (predecessors) of the node. To capture the fanout information into the node embeddings, we enable a bidirectional message-passing mechanism. The concatenated latent node embeddings from the forward and reverse graphs are then translated to nodewise delta-delay predictions using a *teacher sampling* mechanism. With eight possible cell-assignments, the experimental results demonstrate that our model can accurately estimate design-level leakage recovery with an absolute relative error ϵ_{model} under 5.4%. As compared to our previous work, GRA-LPO, we also demonstrate a significant improvement in the model mean squared error.

CCS Concepts: • **Hardware → Physical design (EDA);**

Additional Key Words and Phrases: Discrete gate sizing, directed graph convolution, sequential message passing, leakage optimization

ACM Reference format:
Chung-Kuan Cheng, Chester Holtz, Andrew B. Kahng, Bill Lin, and Uday Mallappa. 2023. DAGSizer: A Directed Graph Convolutional Network Approach to Discrete Gate Sizing of VLSI Graphs. *ACM Trans. Des. Autom. Electron. Syst.* 28, 4, Article 52 (May 2023), 31 pages.
https://doi.org/10.1145/3577019

1 INTRODUCTION

Multi-threshold CMOS provides many tradeoff points between device speed and leakage. These tradeoff points are leveraged during post-layout *swapping* optimizations that reduce leakage power

Authors' address: C.-K. Cheng, C. Holtz, A. B. Kahng, B. Lin, and U. Mallappa, University of California, San Diego, 9500 Gilman Drive, La Jolla, California, 92093, USA; emails: {ckcheng, choltz}@cs.ucsd.edu, {abk, billlin, umallapp}@eng.ucsd.edu.

1084-4309/2023/05-ART52 $15.00
https://doi.org/10.1145/3577019

without sacrificing timing-correctness. During the post-route leakage optimization, *footprint-compatibility* among these tradeoff points offers a great benefit by not introducing any routing disturbances in the layout. Any disturbance to routing involves multiple iterations of *engineering change order (ECO)*-fixing by tools along with manual efforts from routing and verification teams. Figure 1 demonstrates the delay and leakage tradeoff points (normalization to the largest delay and leakage value) in a 28-nm FDSOI foundry enablement. All delay and leakage values are computed for input transition time of 25 ps and output load of 20 fF. The multi-channel-length (P0, P4, P10, P16) and multi-V_{th} (LL and LR) cell variants offer eight footprint compatible swap options [14] (that we refer to as VT1, VT2, VT3, VT4, VT5, VT6, VT7, and VT8). The channel-length variant PX (X = 0, 4, 10, 16) denotes the gate length (channel dimension) biasing value of X (= 0, 4, 10, 16) with respect to the nominal gate length. For example, X = 16 refers to +16 nm biasing as compared to the nominal gate length value. The availability of such fine-grain delay and leakage spectrum between these cell variants is an opportunity that is exploited by the EDA tools for the purpose of leakage optimization.

The complexity of leakage optimization is primarily attributed to two reasons: (1) the number of possible cell reassignments and (2) the constraint that the resulting netlist should not deteriorate the slack of timing paths with negative slack. Figure 2 elaborates on these complexities during the leakage optimization step. The circuit on the top is an example circuit with a timing path (highlighted in red) from FF1 to FF2 having a positive slack of 205 ps. Assume that there are a total of seven cells in the timing path, with an initial assignment of VT1 cell type for all the cells. With eight options for each cell (either stay as VT1 or swap to any of the other seven cell types), a brute-force search for optimal cell assignments must consider 7^8 possibilities. In addition, each cell-swap on the red timing path could also lead to the timing changes of other interacting paths and potentially result in newer timing violations. In this example, reducing the slack below 30 ps leads to newer timing violations on the interacting timing paths. This example serves to illustrate the criticality of contextual awareness of the interacting paths when making cell-level predictions.

To mitigate the intractability of exhaustive search during leakage optimization, state-of-the-art commercial and academic tools use various sensitivity functions to guide iterative cell swapping meta-heuristics [13]. However, such methods are runtime intensive, since any cell swap must be assessed using high-accuracy incremental static timing analysis before being committed. Design methodology teams spend substantial time to develop flows that are likely to achieve best-possible design *power, performance and area (PPA)* metrics within schedule and engineering constraints. However, a design's true PPA quality is known only after a leakage recovery step that is executed by commercial tools such as Cadence Tempus-ECO, Synopsys PrimeTime-ECO or Dorado Tweaker or by internally developed scripts built around incremental STA engines. If designers could achieve accurate predictions of design leakage power without executing the leakage recovery step, then PPA optimizations could be evaluated earlier in the design process with less impact on schedule and tool licensing during the design exploration phase. Moreover, today's physical implementation teams must perform leakage recovery as a signoff step. This is typically done multiple times leading up to tapeout. If the predicted leakage recovery is very small, then designers could choose to skip the leakage recovery step for a given netlist and apply schedule and compute resources elsewhere. This gives rise to a need for an accurate leakage recovery estimator.

An estimate of the recoverable slack per-cell might provide valuable insight about the potential power recovery (based on the available space of swaps). From our experiments, we observe that this per-cell recoverable slack in a design is influenced by various cell-level, path-level, and design-level attributes such as cell types, frequency of the clock, placement utilization, and so on.

Cell-Type	Propagation Delay (ps) | Leakage Power (nW)					
	IVX8		NAND2X3		NOR2X5	
VT1 (LL P0)	0.51	1.00	0.54	1.00	0.48	1.00
VT2 (LR P0)	0.70	0.12	0.68	0.15	0.66	0.13
VT3 (LL P4)	0.58	0.27	0.59	0.27	0.54	0.27
VT4 (LR P4)	0.74	0.04	0.77	0.05	0.75	0.04
VT5 (LL P10)	0.69	0.06	0.69	0.07	0.63	0.07
VT6 (LR P10)	0.90	0.01	0.89	0.02	0.88	0.01
VT7 (LL P16)	0.78	0.03	0.78	0.03	0.72	0.03
VT8 (LR P16)	1.00	0.01	1.00	0.01	1.00	0.01
Max Delta	0.49ps	0.99nW	0.46ps	0.99nW	0.52ps	0.99nW

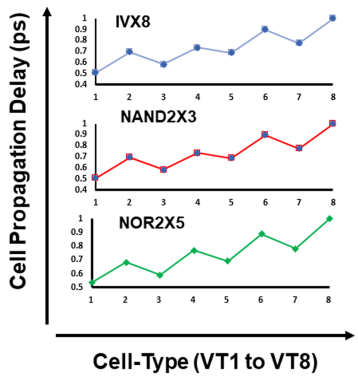


Fig. 1. Normalized propagation delay and leakage power values of an Inverter (INVX8), a two-input NAND gate (NAND2X3) and a two-input NOR gate (NOR2X5) for combinations of V_{th} and channel-length biasing values in 28-nm FDSOI technology. On the right are the corresponding delay plots for various cell types (VT1 to VT8).

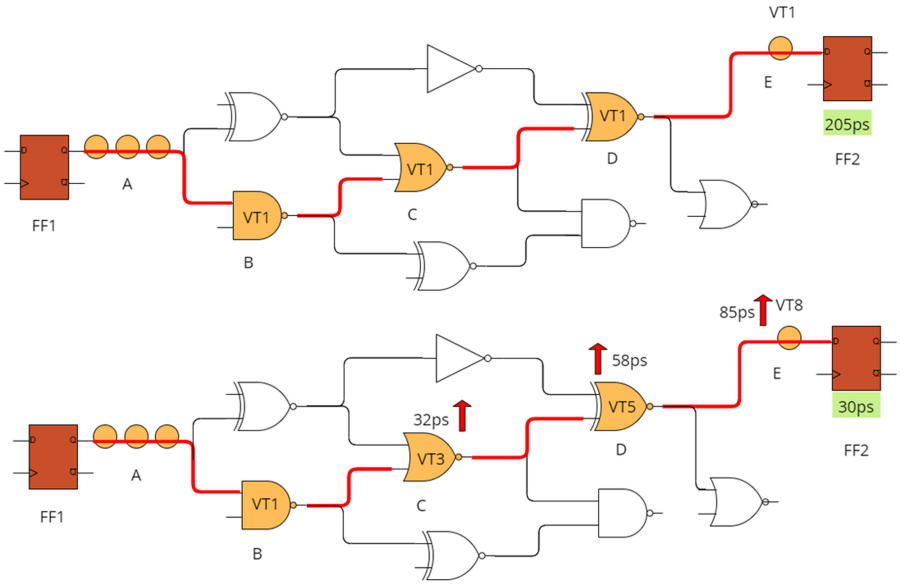


Fig. 2. The circuit on the top represents a pre-recovery netlist with a positive slack (205 ps) for the FF1 to FF2 timing path (red). The bottom circuit represents the post-recovery netlist after appropriate cell-swaps (from the available cell types) by exploiting the positive slack.

For example, a timing path with all VT8 variants (largest delay) cells will not admit leakage power recovery (except possibly through downsizing) even if it has a large magnitude of positive slack. Cell properties such as the depth in a timing path (relative to the path's startpoint), fanin, fanout, and attributes of a cell's sibling cells together determine the amount of available path-slack that of which a cell can make use. In addition, the directed acyclic structure of the netlist plays a crucial role in learning the timing propagation (estimation of arrival times and required times at each node) in the netlist graph. In other words, a leakage recovery model should be aware of

these cell-level attributes and understand the sequence in which timing propagation transpires. This motivates a predictive model that can more clearly interpret directed acyclic timing graphs with nodewise features and learn to predict netlist changes (at the node level) during the recovery process.

In this work, we propose *DAGSizer*, a node-ordering aware sequential message passing mechanism for *directed acyclic graphs (DAGs)*. We apply DAGSizer to the task of discrete *gate-sizing*. The goal of the model is to predict the recoverable slack per-cell (node-level predictions in the graph representation). The predictions are post-processed using library mapping, to generate an ECO netlist along with an estimate of the potential leakage recovery. Importantly, we do not seek to create a new signoff quality gate-sizing tool for leakage recovery, as that function is well served by high-quality commercial and academic tools. Rather, we intend to provide an accurate estimate of the leakage recovery that will result if a specific "golden tool" is launched. The key contributions of our work are summarized as follows:

- **Directed Graphs:** We use graph convolution operations for directed acyclic graphs, which are particularly suitable for the prediction of node assignments during leakage optimization.
- **Sequential Message Passing:** We exploit the topological ordering of the nodes in DAGs and sequentially aggregate the information (message passing) from the direct predecessor-set (immediate parents) of every node. To enable bidirectional message passing, we also use the reversed timing graph in addition to the original timing graph.
- **Teacher Sampling:** Since we derive the intermediate node embeddings sequentially, we use the idea of *teacher sampling* to exploit the predictions made on the predecessor-set, while making predictions on the child nodes. This is in contrast to the previous methods, which generate the node embeddings simultaneously, thereby not capturing the conditional dependency of node-assignments.

The remainder of the article is organized as follows. We provide a brief background to *Graph Convolutional Networks (GCN)* in Section 2. In Section 2.1, we introduce the notations and provide a quick overview of existing GCN-based formulations in the context of discrete gate-sizing. We then motivate the need for a reformulation and introduce our DAGSizer in Section 2.2. Section 3 summarizes the previous works in the category of discrete gate-sizing and GCNs. We provide our mathematical formulation of DAGSizer in Section 4. In Section 5, we discuss our experimental setup and results. In Section 6, we provide insights on the observed results. The last section summarizes our conclusions and future works.

2 GRAPH CONVOLUTIONAL NETWORKS: BACKGROUND

Similarly to the convolutional filters used in regular structures, the convolution operation on graphs involves two main steps (1) weighted aggregation of neighboring node features and (2) applying some activation function (usually a nonlinear) on the aggregated vector, to generate a latent representation of the node, in which the graph connectivity is embedded.

The basic idea of GCN is to aggregate a given node's information with information from its neighboring nodes, while generating a node representation (latent embedding) that comprehends contextual neighborhood information. These latent representations are used for prediction using fully connected layers for various classification or regression tasks. The aggregation operator (convolution) is the key to represent the neighborhood information and is typically realized by parameterized neural networks. Because VLSI netlists can be simplified to directed acyclic graphs, with nodes in a graph representing the cells in the netlist and directed edges representing the pin-pin net connectivity between these cells, GCN-based aggregation serves as a useful means to capture the information of interacting timing paths, while enabling node-level or graph-level predictions.

2.1 Notation

Now, we discuss mathematical notation that we use throughout this article. A graph is represented as $G = (V, E)$, where V is the set of N vertices or nodes and E is the set of edges connecting these nodes.

- The adjacency matrix A representing the graph structure is a $N \times N$ matrix with $A_{ij} = 1$ if $e_{ij} \in E$ and $A_{ij} = 0$ otherwise.
- Each node in the graph can be thought of as having an input feature vector of dimension F, making the feature description X of the graph as a $N \times F$ feature matrix. This initial (extracted from the pre-recovery netlist) representation captures the electrical and physical attributes of a node (cell in the netlist) using a F-dimensional vector for each node x_u for $u \in V$.
- The goal is to generate meaningful high-dimensional (a vector of F' dimensions) node embeddings that capture the graph structure and the feature information from related nodes. As shown in Equation (1), the intermediate node embeddings h_v for each node v are generated using the aggregated embeddings h_u of the neighboring nodes u. To absorb the graph structure that is several hops away from the node, feature aggregation (usually a convolution operator with shared weights) is performed sequentially K times to digest the K-hop information around each node. The superscript l in the equation refers to the index of the convolution layer (or the depth of the convolution layer) and captures the aggregation information l hops away. With this notation, a node's initial representations is $x_u = h_u^0$ and the node's representation at layer $l = 1, 2, \ldots K$ is represented by h_u^l,

$$h_v^{l+1} = \text{Conv}\left(h_u^l \mid u \in \text{Neighbors}(v)\right),\ l = 1, 2, \ldots K. \tag{1}$$

To get a sense of the convolution operator, let us first look at the multiplication of feature vector X and A, which is an aggregation (simple summation) of the node's neighbors and itself. With the introduction of a shared weight vector W of size $F \times F'$, the product AXW can be treated as the weighted aggregation over neighboring nodes and itself. Each column in W indicates the weight contributed by F dimensions of the feature vector, in generating each of the F' dimensions of the node embedding. The output of such a matrix multiplication is an intermediate $N \times F'$ vector representation, which is a linear combination of its neighbors. Typically, such an aggregation is followed by a nonlinear activation f_{non_linear} to extract the power of a universal function approximator,

$$h_v^{l+1} = f_{non\_linear}\left(\text{Conv}\left(h_u^l \mid u \in \text{Neighbors}(v)\right)\right),\ l = 1, 2, \ldots K. \tag{2}$$

In traditional GCN-based formulations of various node prediction problems (including optimization problems associated with VLSI graphs), the intermediate node embedding (latent representations) is limited by the multi-hop local neighborhood aggregation and thus restricted by the depth K of the network. Typically, $K > 3$ does not show any improvement in accuracy of node prediction tasks [28, 32, 33]. The implication is that the models proposed in these works fail to capture the information of the complete timing path. In addition, the directed nature of timing propagation in estimating the node-level arrival time, required time and timing slack is not naturally captured in such latent node representations. As an example, Figure 3 captures predicted post-recovery cell-type distribution of a neighborhood aggregation framework ECO-*graph neural network (GNN)* [32] as compared to the cell distribution from the post-recovery netlist of Tempus-ECO. As shown in the histogram, ECO-GNN fails to produce the correct distribution of classes, confirming our hypothesis about the limitations of local neighborhood-based aggregation methods in complex scenarios (with eight possible cell types).

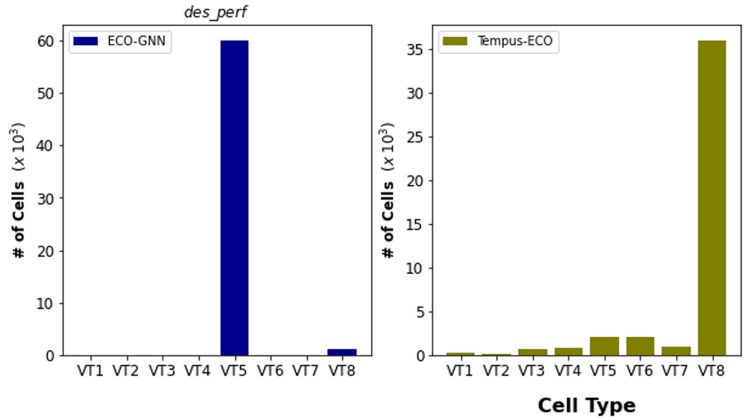


Fig. 3. Predictions from neighborhood-based aggregation (ECO-GNN) as compared to the target response for the *des_perf* design.

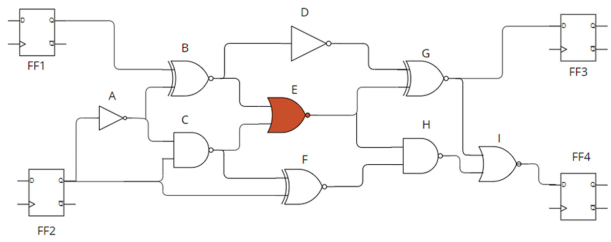
2.2 Graph Convolution for Directed Acyclic Graphs: An Introduction to DAGSizer

Recall that a VLSI circuit (netlist) can be simplified into a DAG $G = (V, E)$. The node-ordering (partial order) of the DAG, characterized by the edge set E has a strong dependency on the timing propagation of various timing paths in the circuit. For node-level classification tasks such as timing and power optimization steps of the physical design flow, we need an intermediate representation of the node that is aware of the topological graph structure. In deriving such an intermediate vector representation, it is important to embed the complete timing path information of the fanin and fanout cone of the node (unlike the previous works that use three-hop neighborhood aggregation). The node-ordering of a DAG allows for updating a node's latent (intermediate) representations based on a message-passing process [25, 44, 46, 48] from the predecessor-set ($u \in$ Predecessors(v) if there is a directed path from node u to node v) sequentially, such that nodes without successors digest the information of the entire graph structure leading to the successor (Equation (3)). We use *depth* or *depth-index* to refer the node ordering-index in the rest of the article. By aggregating feature information from the direct-predecessor set rather than uniformly sampling (direction-agnostic) neighborhoods, we embed the complete information of the timing path that leads to a node. In addition, to embed the timing path information of the successor-set ($u \in$ Successors(v) if there is a directed path from node v to node u), then we use the edge-reversed DAG $G^r = (V^r, E^r)$ to digest the fanout information (Equation (4)) while making node predictions. In the context of natural language processing, this would be explicitly called a *bidirectional recurrent model*. The bidirectional sequential message passing mechanism ensures that the eventual node-level prediction task is aware of the fanin *and* fanout structures of the node.

Pictorial Representation of Node Embedding in DAGSizer: In DAGSizer, one of the core components in deriving the node embedding h_v is the message m_{vf} from its predecessor set (Predecessors(v)). The information carried by this message is an *aggregation* of the embeddings from the parents of the node (h_u). Equations (3) and (4) express such a message aggregation m_{vf} and m_{vr}, for the forward DAG $G = (V, E)$ and the edge-reversed graph $G^r = (V^r, E^r)$, respectively. Here, "Agg" represents the aggregation operation, which will be elaborated in a later section,

$$m_{vf}^{l+1} = \text{Agg}\left(h_u^l \mid u \in \text{Predecessors}(v)\right) \forall v \in V \tag{3}$$

$$m_{vr}^{l+1} = \text{Agg}\left(h_u^l \mid u \in \text{Successors}(v)\right) \forall v \in V. \tag{4}$$



Depth	Nodes	Parent Nodes	Aggregation (Recurrent Message Passing)
0	FF1, FF2	NULL	**NULL**
1	A	{FF2}	$m_A = \text{Agg}(h_{FF2})$
2	B, C	{A, FF1} {A, FF2}	$m_B = \text{Agg}(h_A, h_{FF1})$ $m_C = \text{Agg}(h_A, h_{FF2})$
3	D, E, F	{B}, {B, C}, {C, FF2}	$m_D = Agg(h_B)$ $m_E = Agg(h_B, h_C)$ $m_F = Agg(h_C, h_{FF2})$
4	G, H	{D, E}, {E, F}	$m_G = Agg(h_D, h_E)$ $m_H = Agg(h_E, h_F)$
5	I, FF3	{G, H}, {G}	$m_I = Agg(h_G, h_H)$ $m_{FF3} = Agg(h_G)$
6	FF4	{I}	$m_{FF4} = Agg(h_I)$

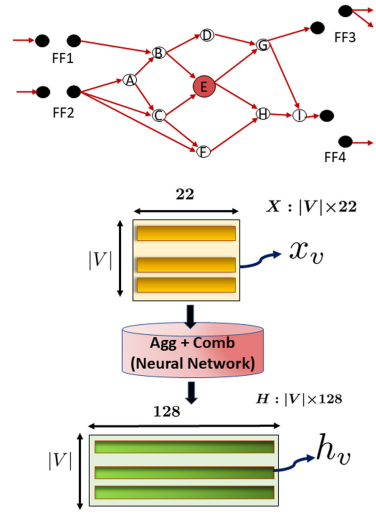


Fig. 4. The graph on top-right is a graphical representation of the circuit (top-left), with nodes as the cells and edges as connecting nets. After topological sorting of the nodes, the node embedding in DAGSizer is generated in an increasing order (from depth 0 to depth 6).

Since the intermediate latent representations of the node should also absorb the feature (self-) information of the node, Equations (5) and (6) introduce the "Comb" operator that combines the message from the parents with the previous representation of v and produces updated representations h_{vf} and h_{vr}. A notable difference from the standard GCN formulation defined in Equation (2) is that the information extracted (m_v^{l+1}) from the parent nodes is from the current layer and not the previous layer. This becomes possible because of processing the nodes in a sequential manner as defined by the DAG ordering. The derived forward (h_{vf}^{l+1}) and reverse (h_{vr}^{l+1}) node embeddings can be computed independently, and the concatenated node embeddings (Equation (7)) serve as the starting point for nodewise predictions. The details of translating the node embeddings to the final nodewise predictions will be explained in Section 4,

$$h_{vf}^{l+1} = \text{Comb}\left\{h_v^l,\ m_{vf}^{l+1}\right\} \forall v \in V, \tag{5}$$

$$h_{vr}^{l+1} = \text{Comb}\left\{h_v^l,\ m_{vr}^{l+1}\right\} \forall v \in V, \tag{6}$$

$$h_v^{l+1} = \left[h_{vf}^{l+1}, h_{vr}^{l+1}\right] \forall v \in V. \tag{7}$$

To perceive the mechanisms of aggregation (Agg) and combine (Comb) operators in a real circuit, Figure 4 shows a simple 13-cell netlist (top-left) with four flops {FF1, FF2, FF3, and FF4} and the combinational logic (consisting of nine cells) between these flops. The corresponding DAG representation (top-right) contains nodes indicating the cells in the netlist and edges representing the pin-pin net connectivity. To recap, nodes in the graph have an initial F-dimensional feature representation that induces $X^{V\times F}$ (the number of nodes is $V = 13$ in this case, and we assume that the number of features is $F = 22$); we seek an intermediate 128-dimensional (F') representation $H^{V\times F'}$ for each node, which embeds the structure of the graph and the feature information from related nodes in the graph. As a post-processing step, we clone the flop-nodes and disconnect the graph at the cloned flop-nodes. The cloning and disconnecting step is performed, because the sequential message passing mechanism of a timing path is required to stop at the endpoints of

the path. Similarly, the message passing mechanism needs to start at the startpoints of the timing paths. This cloning of flops and disconnecting flops also facilitates simultaneous processing of unrelated timing paths. Figure 4 shows cloned and disconnected flop representations of FF1, FF2, FF3, and FF4. This is done to ensure that the message passing gets reset at the endpoint (D pin) of the timing path and restarts at the startpoint (Q pin).

After constructing the DAG, the first step is to topologically sort the nodes (from depth 0 to depth 6). The sequential message passing starts at depth 0 (no parents for FF1 and FF2) and ends at depth 6 (FF3 has a parent G and FF4 has a parent I). To further understand this mechanism, let us look at node E at depth 3. As seen in the graph, node B and node C are the immediate parents of node E. The message m_E that node E gets is an aggregation of hidden representations of node B and node C. This message m_E is combined with node E's feature vector x_E, to generate the hidden representation h_E of the node E. The message passing then proceeds to depth 4, depth 5, and depth 6 sequentially.

3 RELATED WORK

Previous works related to leakage recovery include both continuous and discrete gate sizing optimizations. In the category of continuous sizing, the TILOS work of Fishburn and Dunlop [15] optimizes transistor parameters. The latter perform discrete sizing of standard cells that are characterized by drive strength, input pin capacitance, and other standard library parameters. Typically, combinatorial and/or metaheuristic global optimization approaches are applied, notably Lagrangian relaxation [6, 9, 21, 31, 43, 47], dynamic programming [20, 30, 39], slew budgeting [17], network flow [29], sensitivity-based optimization [19, 40, 45], branch and bound [41], linear programming [5, 7, 23], parallel and randomized algorithms [51], or simulated annealing [42]. All the above-mentioned sizing techniques require up to several tens of hours of runtime to perform leakage recovery for large, complex design blocks; this motivates our quest to find a predictive model. In the category of non-graph learning-based techniques, Derakhshandeh et al. [11] use gate count, gate type, and state-dependent power values for leakage power prediction. Nemani and Najm [36, 37] estimate total power from the RTL netlist, even before gate-level synthesis is performed. They use entropy as a measure of the average activity to be expected in the actual implementation of a circuit, given only its Boolean functional description. The learning-based work [3] proposes a regression model to determine change in timing slack with Vth-swap. Bao [3] highlights two major drawbacks of the methodology, namely, training error associated with complex cells, and the lack of a complete timer graph, which cause large error. Isolating the nodes from its neighborhood does not capture the impact of node-level optimization on timing analysis. Since VLSI circuits can be represented using graphs, any node-level optimization task of the physical design flow needs to comprehend the information from various timing paths that pass through the node, while making these node-level predictions.

GCN algorithms [26] have had proven successes in generalization problems involving arbitrarily structured graphs such as social networks, biological protein structures, and brain structures. The basic idea in GCN is to aggregate information of the neighbor nodes and a given node's self information, while generating a node's latent representation. The non-spectral GCN works such as References [12] and [22] make use of convolution over spatially close neighbors. Variants of non-spectral GCNs such as *Graph Attention Networks (GATs)* [49] are found to be very useful in several node classification and regression problems such as citation networks and protein classification. In GCNs, the aggregation over neighboring nodes is normalized by using the degree of the node as a metric unlike GraphSage [16], where the aggregation over neighboring nodes is not normalized. In GATs [49], the aggregation over node features makes use of the self-attention mechanism (higher attention factor for nodes with similar features).

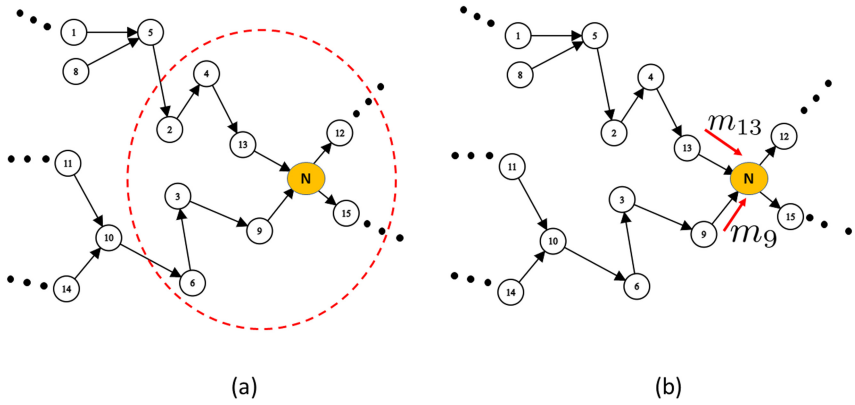


Fig. 5. Representation of GCN's node embedding (here node N) in previous works (a) is restricted by a multi-hop (usually a three-hop) local neighborhood [28, 32, 33] and undirected edges [28, 32], whereas our work is not limited by the depth of the GCN, but uses a sequential message-passing (b) scheme that explicitly exploits the partial ordering of DAG.

Since gate sizing involves node-level optimization (finding an optimal cell type) over timing graphs, graph-based frameworks [28, 32, 33, 35, 50] emerged as an encouraging approach for the task of leakage-recovery prediction. Lee et al. [28] use vanilla GCN-based formulation to predict post-optimization V_{th} assignment probabilities. Their model's classification accuracy is at most 83%, meaning the model mispredicted node assignments for at least 17% of the cells in the design. Similarly, ECO-GNN [32] uses Graph-SAGE inductive learning to formulate the sizing problem as a classification problem and predict node assignment probabilities. In addition to an assumption of undirected edges, both of the works [28, 32] limit feature aggregation to three-hop neighborhoods. To account for directed edges of the timing graphs, GRA-LPO [33] uses GAT-based inductive learning to predict recoverable slack per node and then translates the predictions to leakage recovery by appropriate library mapping. Though edges are directed and the feature aggregation is combined with attention factors, Reference [33] also restricts feature aggregation (with attention factors) to a three-hop neighborhood. Wang and Cao [50] differentiate incoming, outgoing, and sibling neighborhoods by using three separate models for the aggregation of incoming, outgoing, and sibling information. However, as with other relevant work, the aggregation remains limited to three-hop neighborhoods. To embed the complete timing path information into the hidden representation of the node, our proposed DAGSizer exploits the partial ordering of the nodes in a DAG. Figure 5 illustrates a high-level summary of neighborhood aggregation (a) used in the previous works and an improved sequential message passing mechanism (b) of DAGSizer.

4 DAGSIZER: FORMULATION

In this section, we state the problem and more formally introduce the conditional directed graph convolution and sequential message passing operations, the key components of the DAGSizer model. Our method is primarily inspired by DAGNN [48]. Following the notation introduced in Section 2 of Thost and Chen [48] and Section 2 of this work, we describe the DAGSizer model from the *message passing* perspective. A message passing mechanism is composed of three operations: (1) an aggregation operation aggregates a set of incoming *messages*, (2) a combination operation determines an update applied to a node embedding as a function of the node embedding and the incoming messages, and (3) a readout operation composes a set of node embeddings into a subgraph or graph embedding. Note that in this work, we omit (3) due to the fact that gate resizing is a node-level prediction task and graph/subgraph embeddings are not utilized.

4.1 Problem Formulation

Given a pre-recovery netlist in the form of a DAG $G = (V, E)$, where the cells in the netlist are represented by V, and the pin-pin net connectivity between the cells is represented by E. We denote the pre-recovery node features by X^{train}, the adjacency matrix by A^{train}, and nodewise delta-delay values during the optimization by $\tilde{Y}^{train}$. We train a parametric (parameters denoted by Θ) predictive model (that we call DAGSizer) to generate $\hat{Y} = DAGSizer_{\Theta}(X^{train}, A^{train}, \tilde{Y}^{train})$ such that the mean squared loss = $\mathcal{L}(\hat{Y}, \tilde{Y}^{train})$ is minimal and the model parameters can accurately predict the post-recovery delta-delay values of an unseen pre-recovery netlist expressed as $\{X^{test}, A^{test}\}$. Since nodes in the graph represent cells in the netlist, delta-delay of a node is the change in the cell propagation-delay (pre-optimization delay − post-optimization delay) during the gate-sizing optimization task.

4.2 Recurrent Message Passing

Aggregation. Recall that a node's embedding is updated according to an attention-weighted average of the messages produced by its predecessor set. More concretely, given a node v and its associated predecessors $u \in \mathcal{P}(v)$, the incoming messages from each $u \in \mathcal{P}(v)$ are aggregated. We denote the aggregated incoming messages for v at the lth layer of a DAGSizer network m_v^l. m_v^l is computed via the following expression:

$$m_v^l = \text{Agg}^l(h_u^l | u \in \mathcal{P}(v)) = \sum_{u \in \mathcal{P}(v)} \alpha_{vu}^l(h_v^{l-1}, h_u^l) h_u^l. \tag{8}$$

To recap, h_u^l represents the embedding of node u at layer l. Note that α_{vu} is typically parameterized by a small multi-layer perceptron (mlp); $\alpha_{vu}(h_v, h_u) = \text{softmax}_{u_j \in \mathcal{P}(v)}(w_1^\top h_v + w_2^\top h_u)$, where w_1 and w_2 are weights optimized during backpropagation. As References [48] note, edge attributes can be trivially integrated within the attention mechanism. We clarify that the major difference between Equation (8) and the canonical convolutional message passing scheme defined in Equation (2) is that in DAGNN, the aggregation function for v will be only executed after all of its predecessors' latent states have already been computed.

Combination. Given v's incoming message m_v^l, the embedding associated with v at layer l, h_v^l is updated recurrently using m_v^l and the previous representation of node v defined by h_v^{l-1}. Details about the $\hat{y}$ term in the below expression will be explained in Section 4.3,

$$h_v^l = \text{Comb}^l(h_v^{l-1}, m_v^l, \hat{y}). \tag{9}$$

The precise implementation we adopt for the Comb operator is the GRU/LSTM cell [8, 18]. GRU- and LSTM-based networks belong to the family of gated recurrent networks. These methods were originally designed to alleviate issues associated with long-term, specifically variable-length, dependencies (e.g., due to vanishing/exploding gradients during training) [8]. We describe the *modified gated unit* utilized in our framework, characterized by a *forget gate* f^l that governs the tradeoff between the influence of node v's incoming message m_v^l and the influence of the hidden state h_v^{l-1} of node v on h_v^l, conditioned on the labels of the predecessor-set:

$$\begin{aligned}
\tilde{c}^{l-1} &= \sigma(W_c \hat{y} + U_c c^{l-1}) & c^l &= f^l \odot \tilde{c}^{l-1} + i^l \odot \tilde{c}^l \\
f^l &= \sigma(W_f h_v^{l-1} + U_f m_v^l) & \tilde{c}^l &= \phi(W_c h_v^{l-1} + U_c m_v^l) \\
i^l &= \sigma(W_i h_v^{l-1} + U_i m_v^l) & o^l &= \sigma(W_o h_v^{l-1} + U_o m_v^l) \\
h^l &= o^l \odot \phi(c^l), & &
\end{aligned}$$

where σ is a sigmoid function, ϕ is a hyperbolic tangent function, and W and U are parameter weight matrices (learned during backpropagation). At a high level, $\tilde{c}^{l-1}$ and $\tilde{c}^{l}$ respectively correspond to an aggregated embedding of the labels of the parents and previous context and an embedding of the input message of node v with its own embedding. Together, they are used to summarize a "contextual" representation of the timing path—i.e., a memory. f^l and i^l correspond to the input and forget gates. They act as mechanisms that determine how information (derived from the labels of the parents c^{l-1} and the embeddings $\tilde{c}^{l}$) is merged into the updated context-state c^l. o^l corresponds to the "updated state" and c^l is the updated context.[1] A visual depiction of the minimally gated GRU / LSTM cell is provided in Figure 6. Intuitively, this model sequentially learns node embeddings in conjunction with a persistent contextual memory that summarizes the timing path. One comprehensive empirical study in support of GRU-based architectures is conducted in the seminal work [10] of Chung et al., on "challenging sequence modeling task" involving sequences ranging in length from tens (polyphonic music), hundreds (Ubisoft A), and thousands (Ubisoft B). Given the literature and consensus in the ML and NLP communities regarding the efficacy of gated units, we hypothesize that GRU-based architectures are an ideal foundation for timing paths with hundreds of levels. Furthermore, the sequential nature of the model is exploited through the integration of teacher-sampled labels with the context. For more precise technical details, we point the reader to the papers that introduce GRU and LSTM-based models and teacher sampling [4, 8, 18]. Generally, a GRU/LSTM cell has several aspects that differentiate it from a standard layer in a feedforward or convolutional neural network:

(1) It is looped, allowing information to persist through a path (as subsequent elements of the path are observed).
(2) It can control when to let an input influence the computation of the output.
(3) It can control when to remember the output of the previous time step.

4.3 Model Training

The key components of our training framework are subgraph batching, topological sorting, learning intermediate node embeddings, scheduled teacher sampling, and optimization. We present the detailed algorithm in Algorithm 1 and the visual depiction in Figures 7 and 8.

Subgraph Batching: (Step 1 in Figure 7 and line 1 in Algorithm 1). The aggregation operation and intermediate node representations require significant memory and compute resources. Therefore, to account for the limited memory footprint of the GPU while supporting scalability, batching the input netlist graph is necessary. Due to the large size of netlist graphs, we first perform a disjoint cut partitioning. Note that disjointness of each subgraph is a crucial property of nodes that are in the same batch. Canonical methods for batching undirected graphs typically rely on variants of neighborhood sampling [16]. To implement subgraph batching, we adopt the k-way cut clustering implementation of METIS [24] available through PYMETIS [27].

Topological Sorting: (Step 2 in Figure 7 and lines 2 and 3 in Algorithm 1). On each subgraph, we impose an ordering on the nodes. We create this order by topologically sorting of the nodes, done independently for forward and reverse graphs. Formally, given a DAG $G = (V, E)$, a topological sorting of the vertices is given by a linear ordering of vertices such that for all edges $(u, v) \in E$, u precedes v in the ordering. The topological ordering of the nodes facilitates sequential prediction of node labels.

[1]Note that the context at layer l, c^l, depends on hidden states of nodes preceding layer l, and the hidden state of node v, h_v^{l-1}.

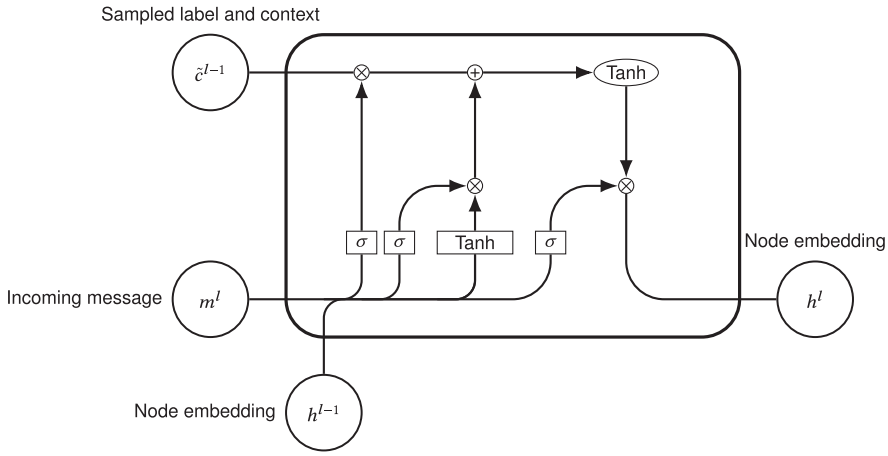


Fig. 6. A visual diagram of a modified gated GRU cell [8].

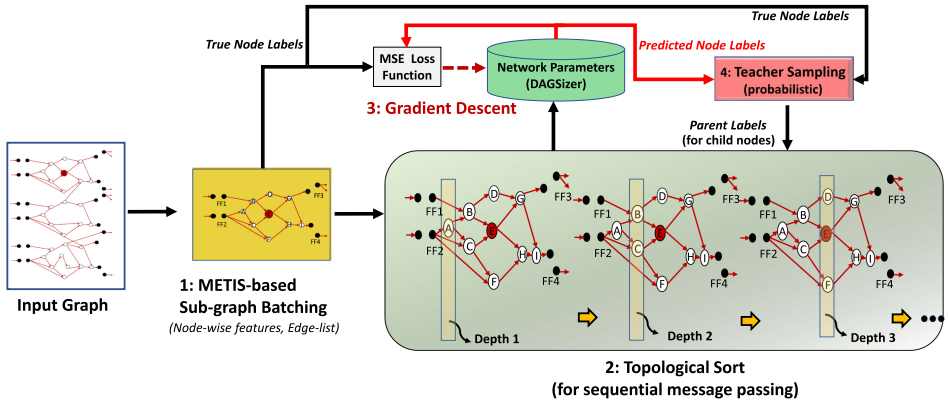


Fig. 7. DAGSizer training framework. Sample preprocessing. Step 1: Subgraph batching. The input graph is decomposed into its independent subgraphs. Step 2: Topological ordering of nodes. Step 3: Network parameters Θ_{t-1} are updated using gradient descent. Step 4: Compute predictions y, utilizing the true label of the parent with probability p_t or (self)-predicted label of the parent with probability $(1-p_t)$ (teacher sampling). p_t starts close to 1 and is decayed over the training process.

Node Embeddings: The node embeddings from the forward graph (line 7 in Algorithm 1) and the reverse graph (line 9 in Algorithm 1) are concatenated to generate node representations (line 10 in Algorithm 1) that comprehends both the fanin and fanout graph structure. With sufficient memory and compute resources, both forward and reverse embeddings can be computed concurrently.

Teacher Sampling: (Step 4 in Figure 7, lines 11–14 in Algorithm 1). After generating the concatenated node representations (line 10 in Algorithm 1), we decode them to generate nodewise labels (line 14 in Algorithm 1). Due to the partial order (implicit for DAGs) imposed on the vertex set, DAGSizer predicts labels sequentially over the circuit. When training hierarchical or structured models such as DAGSizer, an important decision is the granularity of information that is inherited by child nodes from their predecessors. In the vanilla DAGNN model, child nodes receive messages composed of the latent embeddings of their parents. We hypothesize that providing

ALGORITHM 1: 1-layer DAGSizer training

Require: $G_i = (V_i, E_i)$ and node feature=label pairs $V_i = (X_i, y_i)$,
initial parameters ($\mathbf{Encode}_{\Theta_E}, \mathbf{Agg}_{\Theta_{DAG}}, \mathbf{Comb}_{\Theta_{DAG}}, \mathbf{Decode}_{\Theta_D}$).

1: $V_i \leftarrow batchGraph(V_i)$ *partition input graph*
2: $TS_f \leftarrow topsort(G_i)$
ascending order of the forward graph, with key as depth-index and values as nodes
3: $TS_r \leftarrow topsort(G_i^r)$
ascending order of the reverse graph, with key as depth-index and values as nodes
4: $X_{\text{encode}} \leftarrow \mathbf{Encode}_{\Theta_E}(\tilde{X})$ *encode features*
5: **while** Θ not converged **do**
6: **for** each *index* in $TS_f.keys()$ **do**
7: $H_{vf} \leftarrow \mathbf{Comb}_{\Theta_{DAG}}(X_v, \mathbf{Agg}_{\Theta_{DAG}}(X_u | u \in \mathcal{P}(v))), \quad \forall v \in TS_f[index]$
node embeddings in the forward graph
8: **for** each *index* in $TS_r.keys()$ **do**
9: $H_{vr} \leftarrow \mathbf{Comb}_{\Theta_{DAG}}(X_v, \mathbf{Agg}_{\Theta_{DAG}}(X_u | u \in \mathcal{P}(v))), \quad \forall v \in TS_r[index]$
node embeddings in the reverse graph
10: $H_v = [H_{vf}, H_{vr}]$
11: **for** each *index* in $TS_f.keys()$ **do**
12: $\hat{Y}_u^{teach} \leftarrow teacherSampler(\hat{Y}_u)$ *sample parent labels*
13: $H_v^{teach} \leftarrow \mathbf{Comb}_{\Theta_{DAG}}(H_v, \mathbf{Agg}_{\Theta_{DAG}}(H_u, \hat{Y}_u^{teach} | u \in \mathcal{P}(v))), \quad \forall v \in TS_f[index]$
conditional node embeddings
14: Decode labels $\hat{Y}_v \leftarrow \mathbf{Decode}_{\Theta_D}(H_v^{teach})$ *node predictions*
15: Compute loss $l = \mathcal{L}(\hat{Y}, \tilde{Y})$
16: Mask = $\tilde{Y}$.dont_touch
17: Update Θ; $\Theta_{t+1} \leftarrow \text{Adam}(l, \Theta_t, \text{Mask})$

concrete predictions/labels *in addition* to the latent embeddings may facilitate superior learning. However, if the *true label* is solely used during training, then the adoption of generated labels at inference time may lead to poor prediction performance as the model's conditioning context (the sequence of previously generated predictions) diverges from sequences seen during training and prediction errors made at early levels in the graph may cascade to later levels and spoil learning. Scheduled teaching sampling [4] aims to resolve this issue by occasionally (either stochastically or adaptively) providing the *predicted* labels of predecessors as input to child nodes during training. It is important to note that the label of a particular node is not used to make predictions for that node, but only its children. Typically, the true labels associated with nodes are provided exclusively at the start of training (in our case, $p_t = 1$ for the first 30% training epochs). As the model gradually improves its predictive capability, predictions are instead adopted (implemented by decaying of p_t by a factor of 0.9 per epoch). We emphasize that teacher sampling is a train-time augmentation. True labels are stochastically substituted for predictions and *only used during training*. At test time, when labels are unavailable, *predictions* for upstream nodes are used for making predictions on downstream nodes. To maintain consistency with prior work, we impose the same availability of labels as previous work and a similar experimental setup (e.g., ground-truth labels are available during training, but not at test time). To the best of our knowledge, we are the first to propose teacher sampling for node prediction task with directed graph convolution networks.

Training and Backpropagation: (Step 3 in Figure 7, lines 15–17 in Algorithm 1). We formulate the loss function according to standard node-regression principles. Recall that our loss

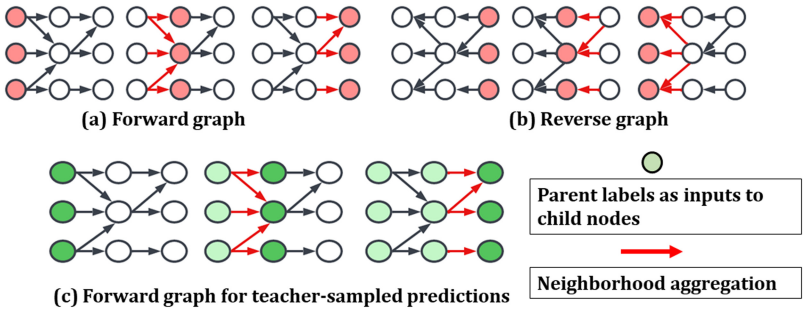


Fig. 8. The node embeddings generated from the forward graph (a), H_{vf} of Algorithm 1, and the reverse graph (b), H_{vr} of Algorithm 1 are concatenated to generate the final node embeddings, $H_v = [H_{vf}, H_{vr}]$ of Algorithm 1. The resulting node embeddings H_v are used for the teacher-sampled decoding phase (c), to predict the node labels. During the teacher-sampled sequential message-passing phase, predictions (light green) from the parents are used in addition to the parent embeddings.

is defined to be the mean-squared-error between the predicted and true delta-delay. Parameters of our framework ($\mathbf{Encode}_{\Theta_E}, \mathbf{Agg}_{\Theta_{DAG}}, \mathbf{Comb}_{\Theta_{DAG}}, \mathbf{Decode}_{\Theta_D}$) are iteratively refined end-to-end via stochastic gradient descent, with gradients computed using the standard backpropagation-through-time algorithm. While updating the model parameters, a key aspect is to ignore the predictions made on the "don't touch cells" (cells that are disabled during the reassignment step of the optimization task).

4.4 Gate Sizing Using DAGSizer

We now extend our training framework to the specific task of Gate Sizing in VLSI netlists. In particular, we summarize the attributes (or features) of nodes that are extracted from the pre-recovery netlist. In addition, we also provide an outline of the various configuration details and the hyperparameters used in our training framework.

4.4.1 Nodewise Features. We evaluate a comprehensive set of 22 node-level features $X^{N\times 22}$ (N is the number of nodes in the graph) that can be extracted from the pre-recovery timing graph. These features are a superset of the features used in previous works [28, 32, 33]. We start with the hypothesis that these 22 features along with net connectivity information (in the form of an edge list) provide sufficient information for the DAGSizer model to learn node-level delay changes during the discrete gate-sizing optimization task. Figure 9 provides a pictorial illustration of the node-level features with reference to node E of our representative timing graph, denoted by $x_E = (f_1^E, f_2^E, \ldots, f_{22}^E)$. The following list in Table 1 summarizes our 22-dimensional feature vector (specific to node E). These features are extracted from the pre-recovery netlist, by our feature extractor. Currently, we do not support MCMM (multi-corner multi-mode) analysis, and the 22 extracted features correspond to a single timing corner. In Table 1, maximum/minimum possible power changes (f_9^E and f_{11}^E) of a node (cell) refer to the maximum/minimum leakage-power change among all possible cell-swaps of the node. Likewise, maximum/minimum delay changes (f_{10}^E and f_{12}^E) refer to the maximum/minimum propagation-delay changes among all possible cell-swaps of the node. To extract f_{10}^E and f_{12}^E from the library file, we use the average of rise and fall delay values, corresponding to the input slew and the output load values from the pre-recovery netlist. In addition to node-level features, we extract pin-pin connections from the netlist and construct the DAG, which is the other input to our framework. While extracting the edge-list, the combinational looping connections are avoided to remove cycles in the generated

Table 1. Nodewise Features Extracted from the Pre-recovery Netlist

Feature Index	Description
f_1^E	worst arrival time of output pins (node E's output pin)
f_2^E	worst slew of input pins (node E's input pins {1,2})
f_3^E	total cap of input pins (node E's input pins {1,2})
f_4^E	load cap of output pin (node E's output net cap and input pin caps of node G {2} and node H {1})
f_5^E	fanout count (number of outgoing edges of node E)
f_6^E	fanin count (number of incoming edges of node E)
f_7^E	worst slack of output pins (node E's output pin)
f_8^E	pre-recovery propagation delay (delay of node E). We use the worst (largest) over all input-output timing arcs and the average of rise and fall delay values.
f_9^E	maximum possible power change (power change of node E)
f_{10}^E	maximum possible delay change (delay change of node E)
f_{11}^E	minimum possible power change (power change of node E)
f_{12}^E	minimum possible delay change (delay change of node E)
f_{13}^E	sensitivity function $\frac{f_9^E}{f_{10}^E}$ of node E
f_{14}^E	sensitivity function $\frac{f_9^E}{f_{10}^E * f_5^E * f_6^E}$ of node E
f_{15}^E	sensitivity function $\frac{f_{11}^E}{f_{12}^E}$ of node E
f_{15}^E	sensitivity function $\frac{f_{11}^E}{f_{12}^E * f_5^E * f_6^E}$ of node E
f_{17}^E	sibling capacitance (input pin cap of node D {2} and node F {1})
f_{18}^E	sibling slack (input pin slack values of node D {2} and node F {1})
f_{19}^E	pre-recovery leakage power of node E
f_{20}^E	worst slew of output pins (worst slew of node E's output pin)
f_{21}^E	total fanin net cap (node E's incoming nets)
f_{22}^E	total fanout slack (slack of node G {2} and node H {1})

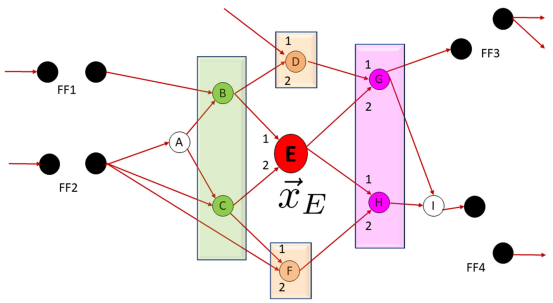


Fig. 9. The node feature vector x_E is a 22-dimensional vector derived from the pre-recovery timing graph.

graph. To ensure that the graph traversal starts at the Q pin of the flop and ends at the D pin of the flop, we make a minor modification to our graph (by including disconnected clones of flop nodes).

4.4.2 Model Configuration. We now describe the high-level configuration details of our model. The DAGSizer framework uses *PyTorch* library to implement encode, decode, aggregation, and combine operations.

Feature Encoder: A linear encoder $\mathbf{Encode}_{\Theta_E}$ is implemented using $torch.Linear(22, 32)$ to translate the 22-dimensional vector to a 32-dimensional vector. The purpose of the initial encoder layer is to learn the relative importance of feature dimensions. We use this feature encoder for all predictive models that we study in Section 5.

Aggregation: A parameterized aggregation operator $\mathbf{Agg}_{\Theta_{DAG}}$ is implemented using the message passing library $torch-geometric.nn.MessagePassing()$, that is used to generate the message vector from the parent nodes. This message vector captures the feature information of the parent nodes and the labels (predicted or true) of the parents (teacher sampling).

Combine: A parametric combine operator $\mathbf{Comb}_{\Theta_{DAG}}$ is used to combine the message vector and the node's feature vector in the forward graph, and generate a 64-dimensional hidden representation of each node. Likewise, the combine operator of the reverse graph generates the other 64-dimensions of the hidden node representation. The concatenation of the two 64-dimensional node representations is used to generate the final 128-dimensional node embedding, i.e., $\mathbf{Comb}_{\Theta_{DAG}} = \{torch.nn.GRUCell(32, 64), torch.nn.GRUCell(32, 64)\}$. For a fair comparison with the previous works, we use 128 dimensions for representing the node embeddings (Equation (2)) of the neighborhood-based aggregation schemes.

Decode: A parametric decode operator translates the hidden vectors of each node to a regression label, i.e., $\mathbf{Decode}_{\Theta_D} = \{torch.nn.Linear(128, 64), torch.nn.ELU, torch.nn.Linear(64, 1), torch.nn.ELU\}$.

Loss Function: The mean-squared loss of node-level delta delay predictions is defined to be

$$\mathcal{L}(\hat{Y}, \tilde{Y}) = \sum_{i=1}^{N} \frac{(\hat{y}_i - \tilde{y}_i)^2}{N} m_i,$$

where $\hat{y}_i \in \hat{Y}$ and $\tilde{y}_i \in \tilde{Y}$. For "don't touch cells" (defined in Section 4.3 as cells that are disabled during the reassignment), we mask the loss (using the m_i flags), implying a masking of the gradients during back propagation. Flops are an example of "don't touch cells" in the leakage optimization step. Because leakage recovery is performed during the signoff stage, the default settings in modern physical design flows recommend the registers to be untouched during the leakage optimization step.

Other Hyperparameters: To be consistent across the predictive models, we use a hidden dimension of 128 to represent intermediate node embeddings and Adam Optimizer with a decaying learning rate: initialized at 0.001 and a decay factor of 1e-5 for every 20 epochs. We use a three-layer convolution for ECO-GNN and GRA-LPO. Since DAGSizer is a sequential message passing aggregation, we use a single hidden layer. To decompose the initial graph (subgraph batching step of Figure 7. and line 1 in Algorithm 1), we adopt the k-way *cut* clustering implementation of METIS [24] via the convenience wrapper PYMETIS [27]. Following best practices [24], we set the number of cut attempts to be one, and the number of iterations to be 10 for all testcases. Crucially, we favor large partitions to avoid unnecessarily splitting timing paths. Since METIS encourages *balanced* partitions, we set the batch-size (number of nodes) according to the available GPU memory and the expected number of nodes in each partition. In general, we select the number of partitions so that batches (subgraphs) consist of roughly 50K nodes. Furthermore, METIS includes a variety of options for seeding graph partitions. The initial partitions may significantly affect the stability of the partitioning procedure. For example, options include spectral cuts, graph growing and greedy graph growing partitions, or Kernighan-Lin-inspired algorithms. The authors of METIS note that the spectral partitioners tend to underperform with respect to speed and quality

Table 2. Criticality of Cut Edges Resulting from Subgraph Batching

Design	Number of Critical/Total Cut Edges	Number of Subgraphs	Number of Nodes
des_perf	0/0	1	61K
b19_fast	42/150	2	89K
vga_lcd	86/5K	2	80K
leon3mp	14/24K	10	503K
netcard	4/28K	11	563K
megaboom	1,576/73K	34	1.7M

compared to graph-growing methods [24]. Of the three graph growing methods, the authors claim that greedy graph growing and "boundary" Kernighan-Lin perform comparatively well. We select greedy graph growing to generate initial partitions for all testcases.

To study the effect of modeling accuracy with and without partitioning, we use the *des_perf* design with 61K nodes and 117K edges, for which the computational graph of can fit into our GPU memory without partitioning the graph. We analyze the accuracy loss and the percentage of cut-edges (w.r.t. the total number of edges in the graph) resulting from partitioning for various batch sizes. Batch size indicates the number of nodes per partition: 50K, 25K, 12K, 6K, 3K, 1K, and 0.5K. For *des_perf*, we observe that *mean squared error (MSE)* stays constant (0.0053) all the way to 0.5K nodes per partition. We believe that there could be three possible reasons for this behavior: (1) the percentage of disconnected edges as compared to the total number of edges is $\leq 2.8\%$ even for a batch size of 500 nodes; (2) cut edges might not always correspond to critical (i.e., having negative slack) timing paths, as suggested by data in 2; and (3) node features (Table 2) such as arrival time, sibling capacitance and sibling slack embed some neighboring information. For the six designs used in our experiments, Figure 10 shows the cut-cost percentages on the y-axis (= percentage of cut-edges w.r.t. the total number of edges in the graph) as a function of the batch size percentage (x-axis). For a batch size of 50K (that can fit into our GPU memory), the cut-cost percentage values (red star on the plots) stays within 2% (y-axis) for all of our designs. Because *des_perf* did not suffer any accuracy loss up to a cut-cost percentage of 3%, even if we can fit the entire graph (moving the red star toward the right) of *megaboom* (or other large graphs) in GPU memory, we believe that the accuracy improvement could be insignificant.

Translation to Sizing Action: Since DAGSizer predicts nodewise delta-delay labels, these labels are translated to the sizing action (among all possible swaps) that most closely matches with the predicted delta-delay value using a simple nearest-neighbor search.

Inference Framework: After learning DAGSizer's parameters (weights) in the training phase (as demonstrated in Figure 7), the inference flow is summarized in Figure 11. The inference flow starts with an input netlist that undergoes DAG translation and feature extraction. We then perform subgraph batching using PYMETIS to decompose the input graph to multiple smaller graphs. The sequential message passing mechanism of the pretrained DAGSizer is applied to each of these subgraphs to predict nodewise delta-delay labels. The generated delta-delay labels are converted to cell types and the changes are rolled back to generate an ECO netlist that can be used for downstream tasks.

5 EXPERIMENTS

To validate the results of our predictive model, we use six designs [1] that were part of the ISPD-13 Gate-sizing contest [38] and the IWLS-05 benchmark suite [2]. The design details such as cell count, flop count, net count, and depth of the logic-cone are shown in Table 3. Our designs

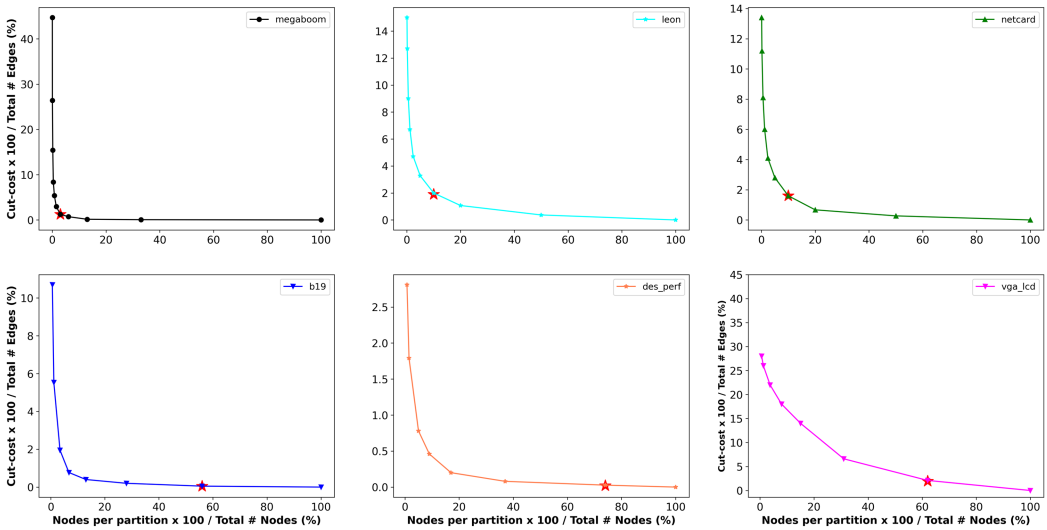


Fig. 10. Plots of cut-cost as a function of various batch sizes. The y-axis indicates the cut-cost (percentage) w.r.t. the total number of edges in a graph and the x-axis represents the nodes per partition (percentage) w.r.t. the total number of nodes in a graph.

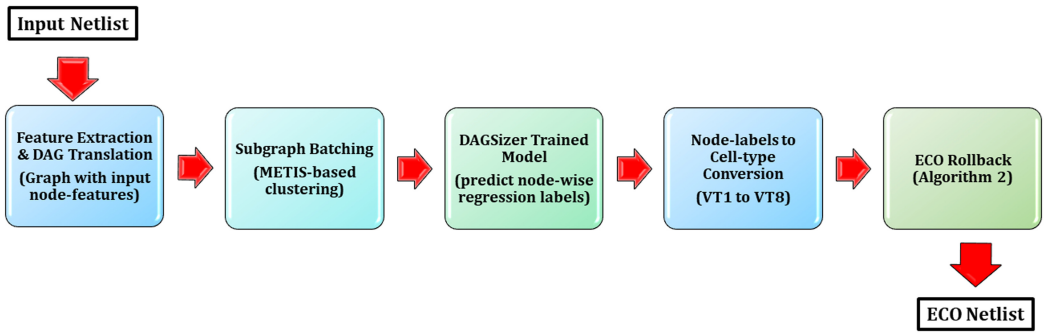


Fig. 11. High-level overview of the DAGSizer inference framework, which uses a pretrained DAGSizer model to generate the ECO netlist.

are synthesized using *Design Compiler* and placed & routed using *Innovus*. The pre-recovery and post-recovery baseline power and timing data is obtained using *Tempus-ECO*. We use a commercial six metal-layer (6lm) BEOL stack 28-nm FDSOI design enablement for the physical design flow. For footprint compatibility in the leakage recovery flow, we use two V_{th} variants (LL, LR) and four channel-length variants ($P0, P4, P10$, and $P16$), making a total of eight possible cell variants. The eventual goal of a predictive model is to accurately estimate the magnitude of a design's potential leakage recovery. Therefore, we compare the design's predicted leakage recovery $\Delta P_{mod} = P^{pre}_{act} - P^{post}_{mod}$, with the golden tool's recovery outcome $\Delta P_{act} = P^{pre}_{act} - P^{post}_{act}$. The relative leakage-recovery error (normalized[2] by the design's pre-recovery leakage power P^{pre}_{act}) given by ϵ_{model} (%) is used to measure the model performance across various designs and

[2]This is in contrast with the normalizing factor ΔP_{act}, used in GRA-LPO.

Table 3. Design Details from the Post-routed Database of Our Benchmarks

Design	Nodes	Edges	Flops	Logic Depth
des_perf	61K	117K	9K	27
vga_lcd	80K	150K	17K	36
b19_fast	89K	271K	7K	55
netcard	503K	1.4M	97K	43
leon3mp	563K	1.5M	108K	40
megaboom	1.7M	4.7M	350K	107

scenarios,

$$\epsilon_{model} = \frac{\Delta P_{mod} - \Delta P_{act}}{P_{act}^{pre}} \; X \; 100\%.$$

5.1 Design Setup

To generate our training and test data, we perform synthesis, place & route, and power-recovery steps at two timing corners *1.10v_tt_125C_rcworst* and *0.90V_ff_125C_rcworst*. For the timing analysis, we use clock period values ranging from 0.7 ns–1.2 ns, based on the complexity of the logic-cone for each design. Our post-route standard cell utilization values are in the range of 75–85%. Similarly to previous works [32, 38], we synthesize and place & route the designs using the fastest VT1 cells (tightest timing constraint). During the ECO phase, each instance is enabled to be swapped with one of the seven types (VT2 to VT8) or can remain unchanged as VT1; thus, there are a total of eight possibilities for each instance reassignment. In reality, the well-bias conflict rule restricts the abutment of LL and LR cells [14]. However, we do not consider the LL and LR isolation rule while generating the train and test data. The short-term goal of our work is to investigate if we can learn the reassignment task in complex scenarios with multiple footprint-compatible sizing options.

Train and Test Data: After place & route, we use the *Tempus* timing tool to perform the timing analysis. This pre-recovery timing graph is used for nodewise feature extraction. As explained in Section 4.4, we extract various node-level features that are essential to the timing propagation and that serve as principal constituents while performing the leakage-recovery task. In addition, we use the pre-recovery database to store the structure of the timing graph as a sparse edge list representation and also record the pre-recovery propagation delay values for each cell in the design. We then perform leakage-recovery using *Tempus-ECO*. While optimizing the design, we provide eight cell types to the tool and therefore the ECO tool exploits the available positive slack to swap and re-assign the cells to one of the eight cell types (VT1 to VT8). In addition to not changing the edge list of the graph, the resulting ECO changes do not lead to any routing disturbances, since the eight available cell variants are footprint compatible. After leakage-recovery, we record the nodewise difference of propagation delay values with respect to the pre-recovery propagation delay values. These nodewise delta-delay values are normalized in the range of [0,1] and then used as the target regression labels.

5.2 Power Recovery on Unseen Designs

Our first experiment validates the accuracy of our predictive models on unseen designs. To evaluate our method, we report a set of validation metrics associated with each design using a one-versus-all strategy (a specific instance of the K-Fold cross validation technique, a standard

Table 4. Leakage Recovery Comparisons with Previous Works (Cross-design Experiment)

Cross-Design Experiments (Power in mW)								
		Golden Tool			Predictive Model			
Design Name	CP (ns)	Total Power	Leakage Power		Post-Recovery Leakage Power (ϵ_{model})			
		Pre-Recovery	Pre-Recovery	Post-Recovery	ECO-GNN [32]	GRA-LPO [33]	DGLPO [50]	DAGSizer
des_perf	1.2	449.4	20.24	9.67	7.61 (−10.1%)	13.83 (20.5%)	8.90 (3.8%)	10.02 (1.7%)
vga_lcd	1.2	144.8	31.99	18.81	17.57 (−3.8%)	26.31 (23.4%)	17.85 (3.0%)	19.26 (1.4%)
b19_fast	0.7	153.8	32.48	13.18	10.07 (−9.5%)	10.92 (−6.9%)	11.24 (5.9%)	12.48 (2.1%)
leon3mp	0.7	1104	220.4	138.4	107.3 (−14.1%)	201.3 (28.5%)	120.2 (8.2%)	145.7 (3.3%)
netcard	0.8	869.4	213.5	132	116.9 (−7.1%)	173.6 (19.5%)	118.2 (6.5%)	140.3 (3.9%)
megaboom	1.2	2802	755	583.9	542.2 (5.5%)	614.2 (3.9%)	551.6 (4.3%)	598.4 (2.0%)

resampling procedure used to evaluate machine learning models). For each design, we train a DAGSizer model on all other designs (for example, when reporting results for *des_perf* we consider a training set comprised of all other designs, excluding *des_perf*). Therefore, we use batches of graphs from five designs and test on the unseen sixth design. We measure pre-recovery leakage power for each design and record the post-recovery leakage power values from *Tempus-ECO*. We compare these actual leakage-recovery values to the model predicted leakage-recovery values. To recap, the difference between the golden post-recovery and pre-recovery leakage power values is defined as ΔP_{act}. For example, if the leakage power values P_{act}^{pre} and P_{act}^{post} for *des_perf* are 20.24 and 9.67 mW, respectively, then the actual recovery (reported by the golden tool) ΔP_{act} is 20.24 mW − 9.67 mW = 10.57 mW. For the same design, DAGSizer predicted leakage recovery is ΔP_{mod} is 20.24 mW − 10.02 mW = 10.22 mW and therefore ϵ_{model} in this case is −0.35/20.24 = −1.7%. We compare the prediction results of DAGSizer with those of our prior work GRA-LPO [33] and a classification-based framework, ECO-GNN [32]. Our implementation of ECO-GNN's model training and inference phases uses Algorithm 1 of Lu et al. [32] and the execution details of their ECO flow are obtained after consultations with the authors. After consulations with the authors of DGLPO, we use Algorithm 1 of Wang and Cao [50] to implement DGLPO. To conduct a fair comparison between the predictive models, we use the same flow (from synthesis to ECO rollback) for all four models (ECO-GNN, GRA-LPO, DGLPO and DAGSizer) except for the nodewise model predictions.[3]

The goal of a predictive model is to make a leakage recovery prediction ΔP_{mod} closer to the actual leakage recovery ΔP_{act}. Since the magnitude of design-level leakage power values have a wide spectrum across various designs, it is reasonable to use the relative prediction error ϵ_{model} (indicated by the % values in parentheses) to compare model performance across various designs. As shown in Table 4, the ECO-GNN makes pessimistic predictions with the absolute relative error ϵ_{model} as high as 14.1%. In the case of GRA-LPO, the absolute value of prediction error goes up to 28.5%. With our DAGSizer formulation, we ensure an absolute relative error under 4% and make more accurate leakage recovery predictions compared to the prior works.

Inference Results: Since our previous work GRA-LPO and the new formulation DAGSizer both use regression formulations to predict the delta-delay values, we compare the MSE during the inference task. Since the node labels are normalized independently for each design to lie in the range of [0,1], the DAGSizer's MSE value of 0.0013 (for *netcard*) corresponds to a mean absolute error of $\sqrt{0.0013} = 0.036 = 3.6\%$. For a raw delta-delay spectrum in the range of [0, 250 ps], this corresponds to an average of 9 ps error in predicting the nodewise delta-delay values. For the same example, GRA-LPO produces an error of 15%, corresponding to 38 ps mean error in predicting the

[3]In addition to the pre-recovery netlist not starting with the tightest-timing cell variant, the experimental settings of GRA-LPO do not enable all the eight cell variants during the leakage optimization process.

Table 5. Inference Statistics of GRA-LPO and DAGSizer for the Cross-design Scenario

Design Name	MSE GRA-LPO [33]	MSE DAGSizer
des_perf	0.0292	0.0090
vga_lcd	0.0410	0.0082
b19_fast	0.0171	0.0031
leon3mp	0.0124	0.0014
netcard	0.0233	0.0013
megaboom	0.0105	0.0044

ALGORITHM 2: ECO Rollback and Timing Analysis

Require: model_eco
dictionary with keys as the depth-index (reverse topological) and values as list of cells

```
Accepted Moves = 0, Total Moves = 0                          reset counters
TS_r = model_eco.keys()                                      get depth indices
for each index in TS_r.keys() do
    cell_list ← model_eco[index]                             cell list at each depth-index
    batch_mode = true                                        enable batch-eco mode
    for each cell in cell_list do
        incr Total Moves
        if slack[cell] > 0 then
            change_cell_to[cell]                             commit the eco change if slack is positive
            incr Accepted Moves
    batch_mode = false                                       disable batch-eco mode
    update_timing                                            update timing before proceeding to the next depth-index
```

raw node labels. As seen in Table 5, DAGSizer converges to lower MSE values as compared to GRA-LPO, suggesting the benefits from our new formulation.

Timing Analysis: In addition to making accurate leakage recovery predictions, we also compare the timing numbers of resulting ECO netlists from the predictive models to the target response. The WNS, number of *Failing Endpoints (FEP)* and percentage of successful moves (Accepted Moves) for the ECO netlist associated with Table 4 are summarized in Table 6. As shown in the Algorithm 2, the predicted ECO changes are loaded depthwise (in reverse topological order). We use the reverse rollback process (starting from endpoints), since it eats up available positive timing slack more slowly than if the rollback were done in forward topological order. At each depth-index, we avoid swapping the cells that are on negative-slack paths (similarly to the approach of Lu et al. [32]). After rolling back the ECO changes at each depth-index, we perform an "update timing" step using the golden incremental timer and use the updated timing results to commit cell changes at the next depth-index. In addition to the timing results, the percentage of accepted moves ($\frac{\text{Accepted Moves}}{\text{Total Moves}} \times 100\%$) is also reported in Table 6. These results indicate the superior performance of the DAGSizer model in terms of the overall timing results (WNS and FEP) and higher percentage of accepted moves (up to 82.4%) compared to previous works (up to 76.4%). When DAGSizer is compared with the timing values from the golden ECO tool, we observe an increase in FEP and a slight degradation in the WNS. However, a majority of these FEPs (for example, 30K of 41K in *netcard*) are in the range of [-25 ps, -0 ps]. slack and can be fixed by tool optimization knobs. The degradation in WNS (as compared to the golden results) across all of the predicted models is a result of

Table 6. Timing Results with the ECO Netlist Generated from Various Predictive Models (Cross-design Experiment)

Cross-design Experiments							
Design	CP (ns)	WNS (ps) / # FEP		WNS (ps) / # FEP / Accepted Moves (%)			
		Golden Pre	Golden Post	ECO-GNN [32]	GRA-LPO [33]	DGLPO [50]	DAGSizer
des_perf	1.2	−42 / 656	−39 / 647	−96 / 3.4K / 70.2%	−95 / 2.9K / 74.0%	−76 / 2.1K / 76.8%	−45 / 1.2K / 81.5%
vga_lcd	1.2	−38 / 3701	−36 / 3696	−98 / 6.2K / 71.0%	−69 / 5.8K / 76.4%	−51 / 4.1K / 78.1%	−41 / 3.3K / 82.4%
b19_fast	0.7	−41 / 10	−41 / 2496	−86 / 7.4K / 68.6%	−62 / 6.5K / 71.2%	−54 / 5.3K / 73.2%	−45 / 4.2K / 76.1%
leon3mp	0.7	−158 / 4	−158 / 77	−362 / 29K / 71.1%	−328 / 21K / 74.8%	−295 / 19K / 76.1%	−205 / 15K / 78.8%
netcard	0.8	−10 / 1064	−10 / 1221	−365 / 53K / 72.5%	−353 / 51K / 75.1%	−210 / 46K / 77.6%	−96 / 41K / 81.9%
megaboom	1.1	−95 / 115K	−100 / 114K	−186 / 148K / 75.1%	−190 / 150K / 74.5%	−174 / 144K / 77.1%	−156 / 135K / 79.5%

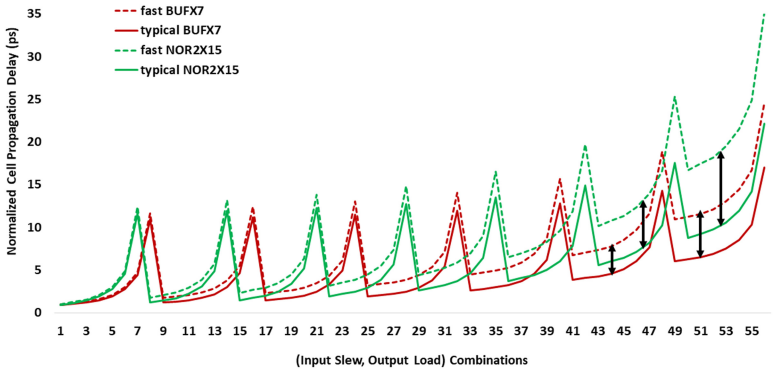


Fig. 12. Normalized propagation-delay (rise) values for two cells (BUFX7 and NOR2X15) in 28-nm FDSOI technology, for 56 combinations of input slew and output load values. These delay values are extracted from 7 × 8 (BUFX7) and 8 × 7 (NOR2X15) Liberty NLDM tables, indexed according to row-major order of table entries.

batched rollback of ECO changes at each depth of the topological ordering. However, the proposed incremental ECO rollback can be avoided by a more accurate model, improvement of the existing modeling approach, and inclusion of much inclusion of much larger and more diverse training data.

5.3 Power-recovery at Unseen Corners

Since the node embeddings capture the timing structure of the graph, we perform a second experiment in which we train the model using the graph derived from the *0.90V_ff_125C_rworst* (fast) timing corner and test on the same design, but using the timing graph and features at an unseen corner *1.10V_tt_125C_rworst* (typical). To extract the timing graph at a timing corner, we perform the complete design implementation (synthesis, place & route, and timing analysis) at the second corner. For 56 (slew, load) combinations of the same timing arc, the plots of rise propagation-delay values of fast (dashed) and typical (bold) timing corners in 28-nm technology are shown in Figure 12. Intuitively, length of the black arrow (delay scaling factor) across various (slew, load) combinations indicates that the relation between these two timing corners is not linear. Furthermore, the variation of the scaling factors across cells (BUFX7 vs. NOR2X15) confirms the non-obvious nature of this correlation. Therefore, we look to a neural network (DAGSizer) to learn the complex correlation between these two timing corners. This is the hypothesis that we investigate in the cross-corner experimental setting. The higher voltage for the typical corner compared to the fast corner is a standard adaptive voltage scaling technique used in modern chips. This voltage scaling is performed to handle the process variations while meeting the power and performance requirements of each device. Our results in Table 7 indicate that the DAGSizer achieves better results

Table 7. Leakage Recovery Comparisons with Previous Works (Cross-corner)

Cross-corner Experiments (Power in mW)								
		Golden Tool			Predictive Model			
Design Name	CP (ns)	Total Power	Leakage Power		Post-Recovery Leakage Power (ϵ_{model})			
		Pre-Recovery	Pre-Recovery	Post-Recovery	ECO-GNN [32]	GRA-LPO [33]	DGLPO [50]	DAGSizer
des_perf	1.2	412.4	9.63	4.78	3.64 (−11.8%)	5.39 (6.3%)	4.18 (6.2%)	4.32 (−4.7%)
vga_lcd	1.2	114.9	16.82	7.33	6.15 (−7.0%)	9.41 (12.3%)	6.44 (5.3%)	8.15 (4.8%)
b19_fast	0.7	142.1	15.25	9.19	10.98 (11.7%)	11.14 (12.7%)	9.91 (−4.7%)	8.35 (5.4%)
leon3mp	0.7	760.2	109.5	31.41	20.34 (−10.1%)	39.56 (7.4%)	25.12 (5.7%)	26.25 (−4.7%)
netcard	0.8	1885	183.9	31.93	21.45 (−5.6%)	40.52 (4.7%)	23.15 (4.8%)	25.90 (−3.3%)
megaboom	1.1	3724	609.1	375.4	332 (−7.12%)	418.6 (6.9%)	404.2 (−4.7%)	398.5 (3.7%)

Table 8. Inference Statistics for the Cross-corner Scenario

Design Name	MSE GRA-LPO [33]	MSE DAGSizer
des_perf	0.0098	0.0053
vga_lcd	0.0157	0.0066
b19_fast	0.0080	0.0059
leon3mp	0.0273	0.0052
netcard	0.0209	0.0043
megaboom	0.0087	0.0034

($|\epsilon_{model}| \leq 5.4\%$) as compared to ECO-GNN ($|\epsilon_{model}| \leq 11.8\%$) and GRA-LPO ($|\epsilon_{model}| \leq 12.7\%$) predictive models.

Inference Results: For the cross-corner experiments, the inference statistics of DAGSizer as compared to GRA-LPO are summarized in Table 8. Recall that the node labels are normalized in [0,1] range. Therefore, an MSE value of 0.0043 (*netcard*) in DAGSizer corresponds to a mean error of $\sqrt{0.0043} = 6.5\%$. If the raw delta-delay spectrum is in the range of $[0, 180$ ps$]$, then this indicates an average of 11.8 ps error in predicting the delta-delay values of nodes. For the same example, GRA-LPO has an error of 14.4%, corresponding to 26 ps mean error in predicting the node labels. Improved MSE values of DAGSizer is a consequence of better interpretation of timing structure context, in the new formulation.

Timing Analysis: The timing results from the resulting ECO changes associated with Table 7 (cross-corner experiments) are summarized in Table 9. With the help of an incremental timer, DAGSizer achieves fewer timing violations (FEP) as compared to the previous works (whose predictions are also applied in tandem with the same incremental timer). Similarly to the cross-design experiments, we follow Algorithm 2 for the rollback process of predicted ECO changes and the timing analysis.

6 DISCUSSION ON RESULTS

6.1 Runtime Statistics

For our model training and inference, we use a *Tesla V100* GPU with 16 GB memory. The DAGSizer framework is developed using *Torch1.10.0* libraries. For the physical design flow, extraction of the timing graph, and the extraction of nodewise features, we use a Xeon Server running at 2.4 GHz. In Table 10, we include both the feature extraction time, data processing and the model inference time. The relative percentages of inference and feature extraction runtimes for DAGSizer and neighborhood-aggregation models (ECO-GNN and GRA-LPO) are shown in Figure 13. Though DAGSizer incurs a sequential overhead in computing the node embeddings, only the nodes belonging to each depth-index and their parents are required to be stored in the GPU memory. Furthermore, as shown in Figure 13, the bulk of the model runtime is invested in feature extraction

Table 9. Timing Results with the ECO Netlist Generated from Various Predictive Models (Cross-corner)

Cross-Corner Experiments							
Design	CP (ns)	WNS (ps) / # FEP		WNS (ps) / # FEP / Accepted Moves (%)			
		Golden Pre	Golden Post	ECO-GNN [32]	GRA-LPO [33]	DGLPO [50]	DAGSizer
des_perf	1.2	−81 / 21	−81 / 917	−148 / 3.1K / 75.3%	−142 / 3.2K / 75.1%	−115 / 2.6K / 78.2%	−92 / 1.8K / 82.0%
vga_lcd	1.2	−45 / 58	−45 / 3K	−152 / 5.6K / 76.8%	−150 / 5.6K / 77.0%	−102 / 4.7K / 79.1%	−84 / 3.1K / 81.2%
b19_fast	0.7	−138 / 17	−138 / 3K	−241 / 6.3K / 72.5%	−232 / 6.1K / 73.4%	−195 / 5.2K / 75.4$	−170 / 4.4K / 78.2%
leon3mp	0.7	−577 / 79	−577 / 4K	−654 / 30K / 75.2%	−664 / 31K / 75.1%	−654 / 29K / 75.8%	−642 / 19K / 79.0%
netcard	0.8	−5 / 90	−5 / 1979	−152 / 42K / 74.0%	−142 / 40K / 76.2%	−136 / 38K / 77.0%	−84 / 32K / 80.1%
megaboom	1.2	−23 / 4K	−23 / 27K	−148 / 61K / 73.8%	−145 / 62K / 73.2%	−128 / 58K / 76.1%	−102 / 54K / 78.8%

Table 10. Inference Runtime (Seconds) for Predictive Models as Compared to Tempus-ECO

Design	Tempus-ECO	ECO-GNN [32]	GRA-LPO [33]	DAGSizer
des_perf	3,577	44.5	51.2	57.8
vga_lcd	3,926	48.2	52.3	59.9
b19_fast	5,092	47.8	56.7	63.2
leon3mp	13,007	270.4	297.4	335.7
netcard	12,749	263.6	286.5	349.3
megaboom	26,250	631.5	640.0	795.2

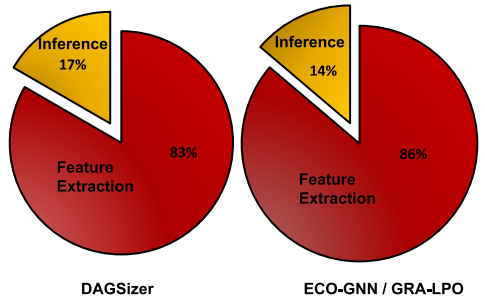


Fig. 13. Average inference and feature extraction percentages (of the total runtime) for DAGSizer and neighborhood aggregation models (ECO-GNN and GRA-LPO).

and therefore the higher model inference time of DAGSizer is not a major concern. In summary, with smaller runtimes when compared to the golden ECO tool, the predictive models serve as quick estimators of the golden tool's leakage recovery.

6.2 Improvement over ECO-GNN

Recall that ECO-GNN is a formulation based on GraphSAGE and predicts probabilities for each of the eight cell types. The neighborhood aggregation scheme of ECO-GNN may not be able to fully understand the intricacies of the timing propagation as it is incapable of modeling the directed nature of timing graphs. In addition, the simultaneous predictions of the nodes could lead to pessimistic predictions. As shown in Figure 14, the resulting cell-variant distributions of the post-recovery designs of ECO-GNN are dominated by a majority cell variant of the ground-truth labels. For *leon3mp* and *netcard* designs, the model predictions are dominated by a single VT8 variant. However, DAGSizer makes more realistic predictions that look closer to the target distribution. A possible explanation for the mismatch of DAGSizer when compared to the target distribution

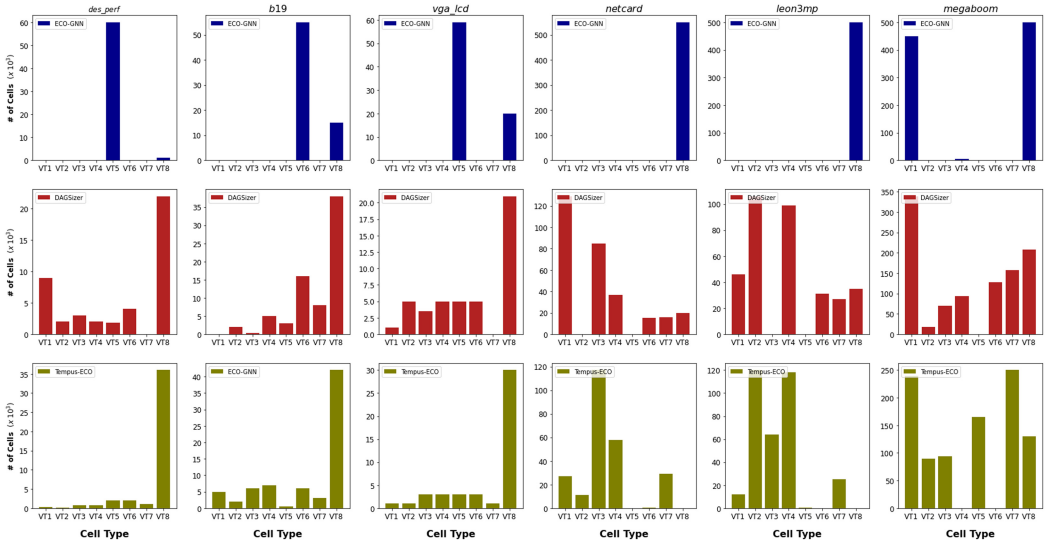


Fig. 14. Predictions from neighborhood-based aggregation ECO-GNN (top) and DAGSizer (middle) compared to the target, from the cross-design experiments.

could be its inability to understand the exact heuristics of the golden ECO tool. Understanding these gaps is part of our ongoing works.

6.3 Improvement over GRA-LPO, ECO-GNN, and DGLPO

Since both GRA-LPO and DAGSizer adopt regression formulations, we evaluate the histogram density of delta-delay predictions in addition to the inference statistics (Tables 5 and 8) in Figure 15. We also include ECO-GNN and DGLPO in the histogram plots, by formulating ECO-GNN and DGLPO as regression tasks, i.e., by removing the softmax layer and using the mean square error loss. This is performed only for the purpose of plots in Figure 15. The x-axis of the plots indicates the normalized prediction error in nodewise delta-delay values, and the y-axis indicates the histogram density (number of nodes). The plots demonstrate that the error region of DAGSizer (green) exhibits a smaller variance—i.e., it is much narrower compared to GRA-LPO (red), ECO-GNN (blue), and DGLPO (yellow). In other words, the accuracy of our model can be interpreted from the narrow error peak of DAGSizer that indicates the zero error region (predicted cell type same as target cell type). Additionally, for some of the designs we see that the error histograms associated with previous works are not centered around zero, implying that the predictions are biased. In contrast, the histograms of DAGSizer exhibit the opposite (desirable, unbiased) behavior. These observations help explain the improved ϵ_{model} of DAGSizer compared to previous works.

6.4 Sensitivity to ECO Engine

To evaluate the dependency of the ECO optimization engine on DAGSizer's predictions, we use a second ECO engine (PrimeTime-ECO) to generate ground truth labels for model training and to measure the model's inference metrics. The scatter plots of Figure 16 show predicted vs. actual leakage recovery values for the Tempus-ECO and PrimeTime-ECO engines, respectively. Predictions closer to the $x = y$ line suggest our model's ability to learn the leakage recovery process using features extracted from the ECO tool's timing graph and its optimization moves.

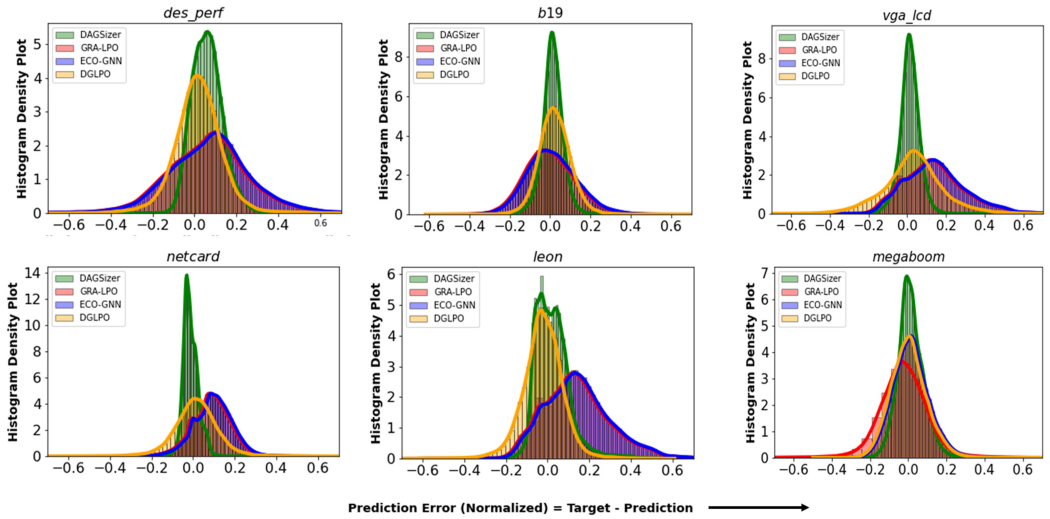


Fig. 15. Histogram plots (scaling factor of 10^3) of normalized error (Target - Prediction = $\tilde{y} - \hat{y}$) for GRA-LPO, ECO-GNN, DGLPO (neighborhood aggregation) and DAGSizer (Recurrent Message Passing) from the cross-design experiments.

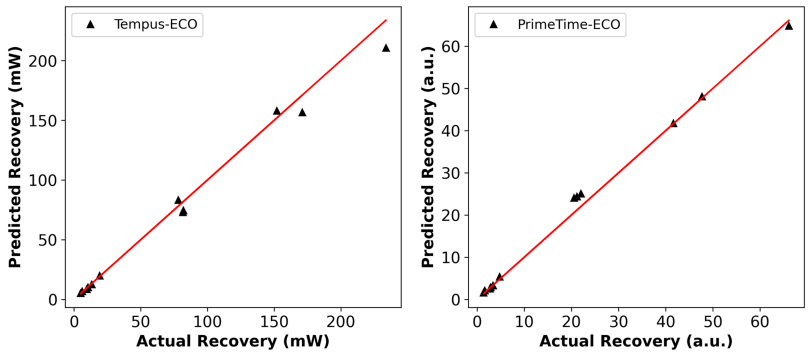


Fig. 16. Scatter plots (predicted vs. actual recovery) showing DAGSizer's ability to learn from different ECO engines. The datapoints on the plot correspond to 12 design-level predictions (six from cross-corner and six for cross-design). To avoid unintended "benchmarking" of the two EDA tools, the plot on the right (PrimeTime-ECO) is scaled to arbitrary units (a.u.).

6.5 Sensitivity to ECO Options

We evaluate DAGSizer's performance with respect to two key knobs of the Tempus-ECO engine. While performing leakage optimization, "*Path Based Analysis (PBA)* effort" and the "target slack" are useful ECO options to balance accuracy and runtime requirements.

PBA vs. GBA: By default, ECO engines perform leakage recovery based on the pessimistic *Graph Based Analysis (GBA)* timing instead of the more realistic PBA timing. Since pessimistic transition-propagation is used for the delay calculations, the GBA mode does not fully utilize the available timing slack while performing leakage recovery. However, the PBA mode propagates the actual transitions (instead of the worst transitions) at each node and recovers more leakage power at the cost of runtime. Figure 17 shows DAGSizer's relative error for the GBA and PBA modes

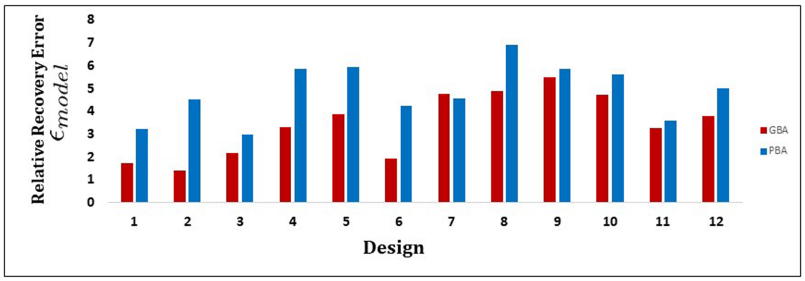


Fig. 17. DAGSizer's GBA vs. PBA relative error (ϵ_{model}) on our designs. The first six indices indicate the cross-design setting and the next six indices indicate cross-corner experiments.

Table 11. Variation of DAGSizer's Relative Error ϵ_{model} for Cross-design Experiments, with Respect to "Target Setup-slack" Option of Tempus-ECO

	Target Setup-Slack (ps)				
Design	0 (baseline)	10	20	30	100
des_perf	1.7%	1.9%	1.9%	2.2%	3.4%
vga_lcd	1.4%	1.4%	1.7%	1.7%	3.2%
b19_fast	2.1%	2.2%	2.1%	2.3%	4.2%
leon3mp	3.3%	3.7%	3.6%	3.4%	5.5%
netcard	3.9%	3.8%	4.1%	3.8%	5.9%
megaboom	2.0%	2.2%	2.2%	2.1%	5.1%

using Tempus-ECO.[4] The first six indices on the x-axis represent six designs (Table 7) in cross-design experimental setting, while indices 7–12 correspond to the cross-corner experimental setting. As seen in the plot, PBA mode incurs more prediction error as compared to the GBA mode. This is because of model's inability to comprehend PBA timing propagation purely from GBA-based node features. This can be mitigated by having another supervised neural network to explicitly model the PBA-GBA correlation at the node or path level.

Target Slack: By default, for the leakage recovery task, ECO engines consider all positive-slack paths and do not impact the negative-slack paths. For cases where the users intend to over-fix the design during optimization, the ECO tools provide an option to add a margin to an existing slack target (0 ps by default). By setting the target slack to a certain threshold value, during the recovery, the tool does not consider paths below this threshold. In Tables 11 and 12, we validate DAGSizer's robustness with respect to the variation of the target setup-slack[5] option. We use four threshold values (10, 20, 30, and 100 ps) and measure the model's relative error for both cross-design (Table 11) and cross-corner (Table 12) experimental settings. The baseline (0 ps) corresponds to the default settings of Tables 4 and 7.

6.6 Benefits of Teacher Sampling

Since DAGSizer is a supervised-learning formulation, we use the target label information only in the training phase. However, our sequential prediction mechanism allows us to condition the

[4]We use `-retime` and `-max_slack 10` options for `set_eco_opt_mode`, to run PBA-based optimization.
[5]We use `-setup_target_slack` option with the `set_eco_opt_mode` command.

Table 12. Variation of DAGSizer's Relative Error ϵ_{model} for Cross-corner Experiments, with Respect to "Target Setup-slack" Option of Tempus-ECO

	Target Setup-Slack (ps)				
Design	0 (baseline)	10	20	30	100
des_perf	−4.7%	−4.7%	−5.1%	−5.4%	−6.5%
vga_lcd	4.8%	5.1%	5.2%	4.9%	6.9%
b19_fast	−5.4%	−5.5%	−5.4%	−5.8%	−7.4%
leon3mp	−4.7%	−4.8%	−4.9%	−4.8%	−7.2%
netcard	−3.3%	−3.4%	−3.5%	−3.6%	−5.8%
megaboom	3.7%	3.7%	3.0%	4.3%	6.1%

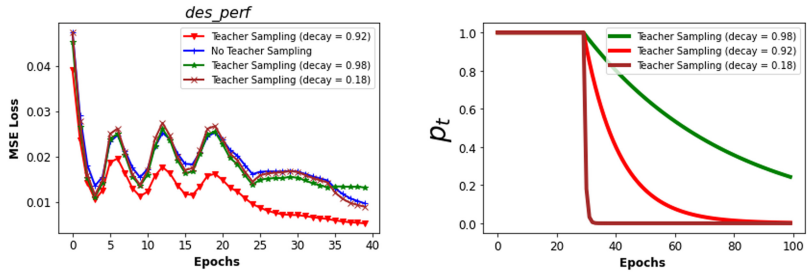


Fig. 18. MSE loss of *des_perf* (cross-design experimental setting) as a function of epochs, for various decay rates of the teacher sampling mechanism.

downstream predictions (that happen later in the topological order of the graph) on the current predictions. During the training process, we stochastically provide the predecessor labels as input to the child nodes; target labels with probability p_t and model predicted labels with probability $1 - p_t$. We use $p_t = 1$ for the first 30% training epochs and then start to decay, to rely more on the model predicted labels (which are used exclusively during testing) and less on the target labels. The plots in Figure 18 indicate superior inference performance with teacher sampling (red) as compared to the blue curve (no label passing from predecessors to child nodes). These plots also corroborate the importance of determining a good sampling probability. For example, a decay factor of 0.98 (green curve) results in the model's over-reliance on the target labels, resulting in poor inference performance. However, a low decay factor of 0.18 (brown curve) also implies poor inference performance because of not sufficiently exploiting the availability of target labels during the training phase. Therefore, we need to balance between overreliance on the training labels (some form of over-fitting) and not reasonably exploiting the training labels (under-fitting). We follow the standard practice of using the validation dataset, to determine an optimal decay factor.

7 CONCLUSIONS

In this work, we propose a DAG partial-ordering aware message-passing mechanism for the discrete gate-sizing problem. Using the timing graph and nodewise feature vectors extracted from the pre-recovery database and the corresponding nodewise delta delay values during leakage optimization, the proposed DAGSizer model learns to predict nodewise delay changes during the leakage recovery optimization for unseen scenarios (designs and corners). Crucially, we demonstrate the necessity of models that are aware of the directed nature of timing graphs. Extensive experiments clearly indicate superior relative recovery prediction error (ϵ_{model}) with lower bias as compared

to the previous predictive models, and under 5.4% absolute relative error for both cross-design and cross-corner experiments. As part of our future work, we seek to explore alternatives that are well suited for scaling sequential message passing to larger graphs, such as the SAR strategy [34].

REFERENCES

[1] OpenCores. Retrieved from https://opencores.org.

[2] IWLS. 2005. IWLS 2005 Benchmarks. Retreived from https://iwls.org/iwls2005/benchmarks.html.

[3] S. Bao. 2010. Optimizing Leakage Power Using Machine Learning. Retrieved from http://cs229.stanford.edu/proj2010/Bao_OptimizingLeakagePowerUsingMachineLearning.pdf.

[4] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled sampling for sequence prediction with recurrent neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems, Volume 1*. MIT Press, Cambridge, MA, 1171–1179.

[5] M. R. C. M. Berkelaar and J. A. G. Jess. 1990. Gate sizing in MOS digital circuits with linear programming. In *Proceedings of the European Design Automation Conference (EDAC'90)*. 217–221. https://doi.org/10.1109/EDAC.1990.136648

[6] Chung-Ping Chen, C. C. N. Chu, and D. F. Wong. 1999. Fast and exact simultaneous gate and wire sizing by Lagrangian relaxation. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 18, 7 (1999), 1014–1025. https://doi.org/10.1109/43.771182

[7] D. G. Chinnery and K. Keutzer. 2005. Linear programming for sizing, V/sub th/ and V/sub dd/ assignment. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'05)*. 149–154. https://doi.org/10.1109/LPE.2005.195505

[8] KyungHyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder-decoder approaches. arXiv:1409.1259. Retrieved from http://arxiv.org/abs/1409.1259.

[9] Hsinwei Chou, Yu-Hao Wang, and Charlie Chung-Ping Chen. 2005. Fast and effective gate-sizing with Multiple-Vt assignment using generalized lagrangian relaxation. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'05)*. Association for Computing Machinery, New York, NY, 381–386. https://doi.org/10.1145/1120725.1120881

[10] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *Proceedings of the Workshop on Deep Learning at the Conference and Workshop on Neural Information Processing Systems (NIPS'14)*.

[11] J. Derakhshandeh, N. Masoumi, S. Aghnoot, B. Kasiri, Y. Farazmand, and Akbarzadeh. 2005. A precise model for leakage power estimation in VLSI circuits. In *Proceedings of the 5th International Workshop on System-on-Chip for Real-Time Applications (IWSOC'05)*. 337–340. https://doi.org/10.1109/IWSOC.2005.23

[12] David K. Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alan Aspuru-Guzik, and Ryan P. Adams. 2015. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett (Eds.), Vol. 28. Curran Associates, Inc.

[13] Hamed Fatemi, Andrew B. Kahng, Hyein Lee, Jiajia Li, and Jose Pineda de Gyvez. 2019. Enhancing sensitivity-based power reduction for an industry IC design context. *VLSI J.* 66 (1 May 2019), 96–111. https://doi.org/10.1016/j.vlsi.2019.01.008

[14] Hamed Fatemi, Andrew B. Kahng, Hyein Lee, and José Pineda de Gyvez. 2020. Heuristic methods for fine-grain exploitation of FDSOI. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 39, 10 (2020), 2860–2871. https://doi.org/10.1109/TCAD.2019.2935053

[15] John P. Fishburn and Alfred E. Dunlop. 1985. TILOS: A posynomial programming approach to transistor sizing. In *Proceedings of the IEEE International Conference on Generalized Anxiety Disorder (GAD'85)*. 326–328.

[16] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc.

[17] Stephan Held. 2009. Gate sizing for large cell-based designs. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition*. 827–832. https://doi.org/10.1109/DATE.2009.5090777

[18] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Comput.* 9, 8 (November 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[19] Jin Hu, Andrew B. Kahng, SeokHyeong Kang, Myung-Chul Kim, and Igor L. Markov. 2012. Sensitivity-guided metaheuristics for accurate discrete gate sizing. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'12)*. Association for Computing Machinery, New York, NY, 233–239. https://doi.org/10.1145/2429384.2429428

[20] Shiyan Hu, Mahesh Ketkar, and Jiang Hu. 2007. Gate sizing for cell library-based designs. In *Proceedings of the 44th ACM/IEEE Design Automation Conference*. 847–852.

[21] Yi-Le Huang, Jiang Hu, and Weiping Shi. 2011. Lagrangian relaxation for gate implementation selection. In *Proceedings of the International Symposium on Physical Design (ISPD'11)*. Association for Computing Machinery, New York, NY, 167–174. https://doi.org/10.1145/1960397.1960436

[22] Ashesh Jain, Amir Roshan Zamir, Silvio Savarese, and Ashutosh Saxena. 2016. Structural-RNN: Deep learning on spatio-temporal graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)* (2016), 5308–5317.

[23] Kwangok Jeong, Andrew B. Kahng, and Hailong Yao. 2009. Revisiting the linear programming framework for leakage power vs. performance optimization. In *Proceedings of the 10th International Symposium on Quality Electronic Design*. 127–134. https://doi.org/10.1109/ISQED.2009.4810282

[24] George Karypis and Vipin Kumar. 1999. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.* 20 (01 1999), 359–392. https://doi.org/10.1137/S1064827595287997

[25] Eliyahu Kiperwasser and Yoav Goldberg. 2016. Easy-first dependency parsing with hierarchical tree LSTMs. *Trans. Assoc. Comput. Ling.* 4 (2016), 445–461. https://doi.org/10.1162/tacl_a_00110

[26] Thomas N. Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *Proceedings of the International Conference on Learning Representations (ICLR'17)*.

[27] A. Klöckner. 2022. PyMetis: A Python Wrapper for METIS. Retrieved from https://github.com/inducer/pymetis.

[28] Wonjae Lee, Yonghwi Kwon, and Youngsoo Shin. 2020. Fast ECO leakage optimization using graph convolutional network. In *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI'20)*. Association for Computing Machinery, New York, NY, 187–192. https://doi.org/10.1145/3386263.3406916

[29] Li Li, Peng Kang, Yinghai Lu, and Hai Zhou. 2012. An efficient algorithm for library-based cell-type selection in high-performance low-power designs. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'12)*. Association for Computing Machinery, New York, NY, 226–232. https://doi.org/10.1145/2429384.2429427

[30] Yifang Liu and Jiang Hu. 2010. A new algorithm for simultaneous gate sizing and threshold voltage assignment. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 29, 2 (2010), 223–234. https://doi.org/10.1109/TCAD.2009.2035575

[31] Vinicius S. Livramento, Chrystian Guth, José Luís Güntzel, and Marcelo O. Johann. 2014. A hybrid technique for discrete gate sizing based on lagrangian relaxation. *ACM Trans. Des. Autom. Electron. Syst.* 19, 4, Article 40 (August 2014), 25 pages. https://doi.org/10.1145/2647956

[32] Yi-Chen Lu, Siddhartha Nath, Sai Surya Kiran Pentapati, and Sung Kyu Lim. 2020. A fast learning-driven signoff power optimization framework. In *Proceedings of the IEEE/ACM International Conference On Computer Aided Design (ICCAD'20)*. 1–9.

[33] Uday Mallappa and Chung-Kuan Cheng. 2021. GRA-LPO: Graph convolution based leakage power optimization. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference (ASP-DAC'21)*. 697–702.

[34] Hesham Mostafa. 2022. Sequential aggregation and rematerialization: Distributed full-batch training of graph neural networks on large graphs. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 265–275.

[35] S. Nath et al. 2022. Invited: Generative self-supervised learning for gate sizing. In *Proceedings of the Design Automation Conference (DAC'22)*. 1–4.

[36] M. Nemani and F. N. Najm. 1996. Towards a high-level power estimation capability [digital ICs]. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 15, 6 (1996), 588–598. https://doi.org/10.1109/43.503929

[37] M. Nemani and F. N. Najm. 1999. High-level area and power estimation for VLSI circuits. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 18, 6 (1999), 697–713. https://doi.org/10.1109/43.766722

[38] Muhammet Mustafa Ozdal, Chirayu Amin, Andrey Ayupov, Steven Burns, Gustavo Wilke, and Cheng Zhuo. 2012. The ISPD-2012 discrete cell sizing contest and benchmark suite. In *Proceedings of the ACM International Symposium on International Symposium on Physical Design (ISPD'12)*. Association for Computing Machinery, New York, NY, 161–164. https://doi.org/10.1145/2160916.2160950

[39] Muhammet Mustafa Ozdal, Steven Burns, and Jiang Hu. 2011. Gate sizing and device technology selection algorithms for high-performance industrial designs. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'11)*. 724–731. https://doi.org/10.1109/ICCAD.2011.6105409

[40] Mohammad Rahman and Carl Sechen. 2012. Post-synthesis leakage power minimization. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'12)*. 99–104. https://doi.org/10.1109/DATE.2012.6176440

[41] Mohammad Rahman, Hiran Tennakoon, and Carl Sechen. 2011. Power reduction via near-optimal library-based cell-size selection. In *Proceedings of the Design, Automation Test in Europe*. 1–4. https://doi.org/10.1109/DATE.2011.5763293

[42] Tiago Reimann, Gracieli Posser, Guilherme Flach, Marcelo Johann, and Ricardo Reis. 2013. Simultaneous gate sizing and Vt assignment using Fanin/Fanout ratio and simulated annealing. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'13)*. 2549–2552. https://doi.org/10.1109/ISCAS.2013.6572398

[43] Subhendu Roy, Derong Liu, Junhyung Um, and David Z. Pan. 2015. OSFA: A new paradigm of gate-sizing for power/performance optimizations under multiple operating conditions. In *Proceedings of the 52nd ACM/EDAC/IEEE Design Automation Conference (DAC'15)*. 1–6. https://doi.org/10.1145/2744769.2744885

[44] Bing Shuai, Zhen Zuo, Bing Wang, and G. Wang. 2016. DAG-recurrent neural networks for scene labeling. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*. 3620–3629.

[45] Ashish Srivastava, Dennis Sylvester, and David Blaauw. 2004. Power minimization using simultaneous gate sizing, dual-vdd and dual-vth assignment. In *Proceedings of the 41st Annual Design Automation Conference (DAC'04)*. Association for Computing Machinery, New York, NY, 783–787. https://doi.org/10.1145/996566.996777

[46] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, 1556–1566. https://doi.org/10.3115/v1/P15-1150

[47] H. Tennakoon and C. Sechen. 2002. Gate sizing using Lagrangian relaxation combined with a fast gradient-based pre-processing step. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*. 395–402. https://doi.org/10.1109/ICCAD.2002.1167564

[48] Veronika Thost and Jie Chen. 2021. Directed acyclic graph neural networks. In *International Conference on Learning Representations*.

[49] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph attention networks. In *International Conference on Learning Representations*.

[50] Kai Wang and Peng Cao. 2022. A graph neural network method for fast ECO leakage power optimization. In *Proceedings of the 27th Asia and South Pacific Design Automation Conference (ASP-DAC'22)*. 196–201.

[51] Tai-Hsuan Wu and Azadeh Davoodi. 2009. PaRS: Parallel and near-optimal grid-based cell sizing for library-based design. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 28, 11 (2009), 1666–1678. https://doi.org/10.1109/TCAD.2009.2028682

Received 3 May 2022; revised 31 October 2022; accepted 10 December 2022