LSTP : A Logic Synthesis Timing Predictor

Haisheng Zheng[1] Zhuolun He[1,2] Fangzhou Liu[1,2] Zehua Pei[1,2] Bei Yu[2]
[1]Shanghai AI Laboratory, Shanghai, China
[2]The Chinese University of Hong Kong

***Abstract*—The ever-growing complexity of modern VLSI circuits brings about a substantial increase in the design cycle. As for logic synthesis, how to efficiently obtain physical characteristics of a design for subsequent design space exploration emerges as a critical issue. In this paper, we propose LSTP , an ML-based logic synthesis predictor, which can rapidly predict the post-synthesis timing of a broad range of circuit designs. Specifically, we explicitly take optimization sequences into consideration so that we can comprehend the synergy between optimization passes and their effects on netlists. Experimental results demonstrate that we outperform state-of-the-art remarkably.**

I. Introduction

The last decades have seen tremendous progress in logic synthesis, which provides a dramatic productivity boost in digital circuit design [1]. A typical objective of logic synthesis is to find an implementation of a boolean function f that minimizes area and delay, serving as an essential instrument to push the limits of performance and power consumption [2]. Logic synthesis is critical: firstly, architecture exploration relies on the acquisition of metrics reported by logic synthesis [3]; secondly, logic synthesis quality determines the best possible design space of subsequent procedures [4]. The facts above also indicate that logic synthesis might be performed for many times in the design flow. Therefore, the *predictability* [5] of logic synthesis becomes an issue of common interest: can we efficiently predict the desired metrics without actually running expensive logic synthesis?

The ever-growing complexity of integrated circuit designs adds to the time consumption of logic synthesis, whereas Moore's law implies an exponential increase in design time with technological advancement. Massive efforts have been put into improving logic synthesis efficiency, such as a *divide-and-conquer* framework that synthesizes a design in a modular fashion [4], heuristic methods for logic minimization [6], [7], enhancing *satisfiability* (SAT)-based modeling and solver efficiency [8], [9], and parallel acceleration through modern computing platforms like GPUs [10], [11]. These approaches have successfully obtained orders of magnitude speedup and thus increased the usability of modern logic synthesis tools; however, they are still one step away from our goal of acquiring desired metrics without even launching the synthesis.

In order to obtain the physical characteristics of a circuit quickly, numerous previous works utilized machine learning (ML) to predict the properties of circuits, such as timing, power, area, and so on. D-SAGE [12] proposes an enhanced graph neural network (GNN) model to learn the operation mapping pattern in high-level synthesis (HLS), which is further exploited for HLS delay estimation. Yu *et al.* [13] employ a long short-term memory network (LSTM) to achieve accurate estimation of sequential behaviors and predicts delay & area by a transfer learning scheme after the synthesis flow. PowerNet [14] is a convolutional neural network (CNN)-based approach for dynamic IR drop prediction. GRANNITE [15] uses a GNN model to predict the circuit's power consumption. De *et al.* [16] present comprehensive machine learning methods for HLS delay estimation. Deep H-GCN [17] performs analog circuit degradation prediction (i.e., aging) through heterogeneous GNNs. LOSTIN [18] predicts synthesis flow performance by combining spatial (netlist) and temporal (flow) information encoded by GNN and LSTM. By leveraging the blossoms of machine learning, especially deep learning methods, efficient circuit metric acquisition becomes possible and popular. We argue that a key reason for a successful application of ML to metric prediction is to select a proper model to capture different kinds of accessible raw information. Among the above works, graph neural networks find prosperous uses [12], [15], [17], as they are a proper fit to model element connection and topology, such as for netlist representation learning. On the other hand, CNNs are suitable for layout-based tasks [14], while sequential structures like LSTMs are designed for series modeling [13].

Moreover, logic synthesis recipes are not one-size-fits-all. Recently, OpenABC-D [19] has pointed out quantitatively that the similarity between the best synthesis recipes for a set of benchmark circuits is less than 30%. In other words, the optimal optimization sequence in logic synthesis is design-dependent. However, this sequence used in logic synthesis tools either relies on the settings provided by the developer, or is customized by end-users. Neither of the two options is ideal: Design-dependent optimal sequences cannot be determined at the time of tool development; end-users are even impossible to understand the effects of the sequences, let alone decide which one to use. There are a few recent research works related to optimization sequence quality improvement. Yu *et al.* [20] propose to train a CNN to predict the quality of an optimization sequence, and randomly samples from the sequence design space to look for better sequences. This approach inherently ignores the orchestration between netlist and optimization sequence, which is not desired. Besides, if 'good' sequences are sparse in the whole design space, random sampling may not generate good enough candidates for the model. Reinforcement learning is leveraged [21], [22] to generate fixed-length optimization sequences. In their common settings, the state space encodes current progress (viz. current netlist) after applying previous optimization passes to the input netlist, and the action space is to predict the next optimization pass. Although performance gains are reported, these methods

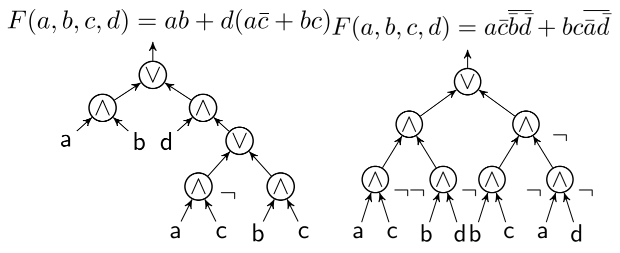

$F(a,b,c,d) = ab + d(a\bar{c} + bc)$
$F(a,b,c,d) = a\bar{c}\overline{\bar{b}\bar{d}} + bc\overline{\bar{a}\bar{d}}$
a
b
d
a
c
b
c
a
c
b
d
b
c
a
d
(a)
(b)

Fig. 1 Two different AIGs for a Boolean function.

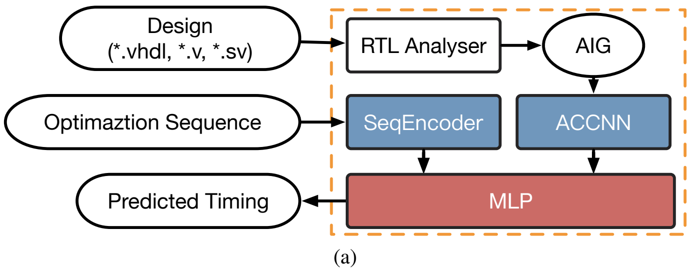


Fig. 2 LSTP Prediction Flow.

are unable to directly give circuit metrics prediction, which is of our interest as introduced.

In this work, we propose LSTP, a machine learning driven logic synthesis timing predictor. LSTP aims to model the complex interaction between optimization passes and their effects on the netlist through an end-to-end trainable neural framework. Specifically, a dedicated *Cascaed-Cone-based* graph neural network is designed to encode the netlist, while an attention mechanism is employed to encode the optimization sequence. The outputs from both models are concatenated and fed to an MLP to predict the post-synthesis timing. In this way, the model effectively comprehends the optimization sequence and its effect on the input netlist. Our major contributions are summarized as follows:

- We present LSTP, a highly efficient machine learning driven logic synthesis timing predictor;
- We propose a novel learning framework that models the interaction between logic synthesis optimization passes and their effects on a netlist;
- We conducted comprehensive experiments on real-world circuit designs to demonstrate that LSTP outperforms state-of-the-art logic synthesis predictors in timing prediction accuracy;
- We further use the timing prediction from LSTP to guide optimization sequence generation, which obtains circuit designs with better actual performance.

II. Preliminaries

Logic synthesis transforms a high-level hardware description (e.g., Verilog, VHDL) into a gate-level netlist after optimizing Boolean functions. It mainly consists of three critical steps, logic optimizations, technology mapping, and post-mapping optimizations. Typically, a gate-level netlist is a Boolean function with binary-valued inputs, and it uses various logical operations such as AND, XOR, and NOT. The And-Inverter-Graph (AIG) and Major-Inverter Graph (MIG) are common representations of this netlist, which allow structural optimizations. The state-of-art synthesis tools first apply a sequence of logic minimization heuristics to transform the circuit in these representations, which is often called the *synthesis transformation*. Every transformation focuses on simplifying the node representation and refactoring Boolean formulas in order to meet an expected area and delay overhead. The *synthesis recipe* consists of many *synthesis transformations* in some order, which plays an important role in the quality of results (QoR).

An And-Inverter-Graph (AIG) is a Directed Acyclic Graph (DAG) used to represent arbitrary Boolean formulas and circuits [23]. An AIG consists of two-input nodes (AND function), dotted edges (NOT function), and terminal nodes (Primary inputs and constants). The construction of AIGs only requires simple AND gates and inverters to express every essential logic operation. Compared with the binary decision diagram (BDD) and the disjunctive normal form (DNF), AIG is in a non-canonical form with no unique Boolean formula representation. This allows for better scalability of circuits represented in AIG form without many formal constraints on circuit hierarchy. The two AIGs in Fig. 1 represent the same logical expression, one with 6 nodes and 4 levels while the other with 7 nodes and 3 levels.

The state-of-the-art open-source logic synthesis tool ABC [24] uses a combination of random/guided simulation of AIGs and Boolean satisfiability (SAT) to improve synthesis performance. Meanwhile, it proposes several AIGs optimization methods to work in synthesis, making it delay-aware and then adjusted the structure of the AIG to match. Typically, combinations/ordering of these optimization passes affect the final circuit performance. However, modeling such effects is not straightforward.

We formulate our timing prediction problem as follows.

Problem 1 (Logic Synthesis Timing Prediction)**.** *Given a gate-level netlist as an And-Inverter Graph (AIG) representing a set of Boolean functions and a sequence of subgraph optimization procedures for the AIG graph, design a novel learning methodology that automatically predicts the final timing after applying the optimization procedures to the AIG.*

III. Algorithms

A. Overall Flow of LSTP

The overall flow of the proposed LSTP is illustrated in Fig. 2. Given an RTL Verilog design and the corresponding optimization sequence as input, a neural model is trained to mimic the behavior of the logic synthesizer and predict the final timing performance. LSTP consists of four stages:

1) **RTL-Analyzer** compiles the input design and transforms it into an And-Inverter-Graph (AIG) representation.
2) **ACCNN** is a trained graph neural network (GNN) for node sampling and feature extraction of the AIG circuit.
3) **SeqEncoder** is a trained Transformer encoder for optimization sequence features extraction.

4) **MLP** aggregates both the optimization sequence features and the circuit diagram features to predict the timing of the input design.

B. RTL-Analyser: Turning Design into AIG

And-Inverter Graph (AIG) is an efficient data structure for the manipulation of large sequential Boolean networks in a variety of applications, including synthesis, technology mapping, and formal verification. In this work, we convert the design RTL into AIG for circuit representation.

Given the hardware description language (HDL) description of the design circuit as input, Yosys [25], an open-source synthesis suite that contains a complete toolchain for HDL source code compilation and gate-level synthesis, is used to parse and compile the input circuit HDL and generate an intermediate representation of the circuit; the intermediate circuit representation is further converted into an AIG representation by an open-source logic synthesis tool.

The circuit is then represented as a directed acyclic graph (DAG) $\mathcal{G}(\mathcal{V},\mathcal{E})$ using the netlist analyzer, denoting vertex set by $\mathcal{V}$ and edge set by $\mathcal{E}$. Vertex $v \in \mathcal{V}$ represents the 2-input gate, while $e \in \mathcal{E}$ denotes either inverters or buffers. Furthermore, we define node type and the number of incoming inverted edges (from primary input to itself) as two node-based features and the edge type as the edge feature.

C. ACCNN: Netlist Feature Extraction

The delay of a circuit depends on the number of hops on the longest path from the PIs to the POs. Intuitively, the longest path of an AIG can be extracted for timing prediction. However, after the *RTL-to-AIG* conversion, two important operations, namely technology mapping and logic optimization, will precede. These two operations will modify the local pattern, number of nodes, connectivity, and topology of the AIG. Thus, predicting timing on the longest path without consideration of the above two operations is inaccurate.

To overcome the above concerns, we wish to design an algorithm to effectively exploit the characteristics of graph representation for the longest path in AIG.

Since the design circuit can be represented as a directed acyclic graph (DAG), and several prior works use GNN to learn netlist representation to yield excellent results [15] [26] [27] [28], we propose to use GNN to learn the representations for the AIG.

GNN encodes nodes as low-dimensional vectors and preserves the structural information of the graph to the maximum extent. By aggregating node and edge features and sharing features with neighbor nodes by message passing, both local and global information of the graph representation can be learned and exploited.

Notwithstanding the structural complexity and variant size of AIGs, we observe that a substantial number of paths with similar topological structures exist in AIG, which contains redundant information. Therefore, we aim to devise a lightweight sampling algorithm to replace the exhaustive adjacent aggregation. GraphSAINT [29] precisely fulfills this purpose by employing several lightweight graph sampling techniques, such as random node sampling, random edge sampling, and random walks. It samples subgraphs from the complete training graph to construct mini-batches, instead of previous approaches that involved node/edge sampling at each layer of GCN. Drawing inspiration from GraphSAINT, we propose a novel network called *Asynchronous Cascaed-Cone* Neural Network (*ACCNN*) to sample and learn the intrinsic features of circuit AIG.

Algorithm 1 Random Sampling for ACCNN

```
Require: AIG $G\langle\mathcal{V},\mathcal{E}\rangle$, sample size $N$ for a PO
Ensure: A list of sampled cascaded cones $L$
  for each $PO \in G$ do
    for $n \leftarrow 1 \cdots N$ do
      $CC \leftarrow$ SampleCascadedCones($G, PO$)
      $L \leftarrow L \cup \{CC\}$
    end for
  end for
  function SampleCascadedCones($G$, $PO$)
    $CC \leftarrow \{PO\}$
    $v \leftarrow PO$
    while $v$ is not a PI do
      $nodes \leftarrow$ predecessors of $v$
      $CC \leftarrow CC \cup nodes$        ▷ include the cone in $CC$
      $v \leftarrow$ a random node in nodes
    end while
    return $CC$
  end function
```

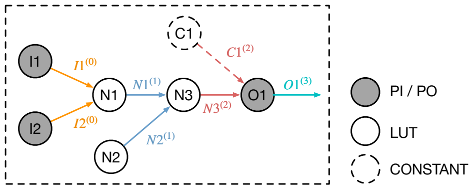
Cascaded Cone. As shown in Algorithm 1, we introduce a random walk-based approach to sample cascaded cones within the circuit randomly. The main focus is on 'paths' that originate from primary inputs (PI) and end at primary outputs (PO). To handle flip-flops (FF) in the circuit, their inputs are considered as POs, and their outputs are treated as PIs.

The sampled cascaded cones play a crucial role in capturing the circuit's behavior for a single clock cycle, thereby enabling precise timing predictions for the entire design through LSTP. We represent the AIG as a graph $G = \langle\mathcal{V},\mathcal{E}\rangle$, where $\mathcal{V}$ represents the set of nodes and E is the set of edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. These sets are provided as inputs to the algorithm.

Moreover, to control the sampling process, we introduce a parameter N, which determines the number of paths to be sampled for each primary output (PO). In our work, *ACCNN* uses N = 4. Sampling more paths does not improve the accuracy of ACCNN and brings additional computational overhead.

Asynchronous. After randomly sampling the cones, our objective is to find a model that effectively summarizes and extracts features from these sampled structures. We aim for a model that resembles logic simulation, efficiently propagating information step-by-step along the sampled paths. *ABGNN* [26] serves this purpose well, which employs an asynchronous message passing scheme similar to the Chandy-Misra-Bryant (CMB) distributed time algorithm [30] used in logic simulation.

Specifically, instead of simultaneously propagating information across all edges in each iteration like synchronous GNNs, the asynchronous message passing scheme in *ABGNN* allows



	I1	I2	N1	N2	N3	C1	O1
T = 0	✓	✓					
T = 1			✓	✓			
T = 2					✓	✓	
T = 3							✓

Fig. 3 A visual illustration of *ACCNN*.

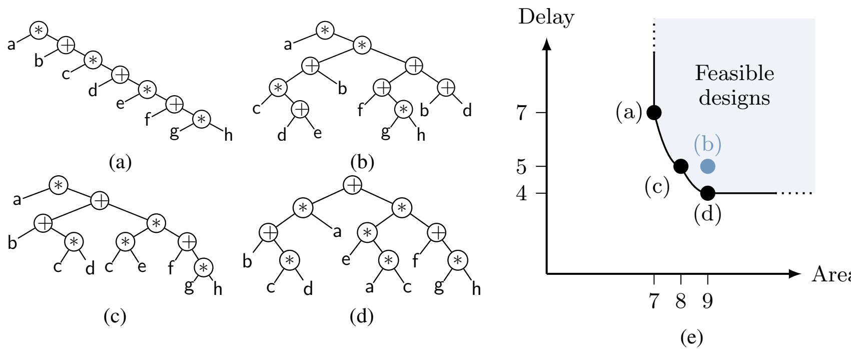


Fig. 4 (a)–(d) Equivalent factored forms; (e) Area/delay trade-off for the trees.

vertices to communicate following the partial order induced by the DAG structure. Messages originate from the leaf vertices and flow towards the target vertex v following the topological ordering. At the k^{th} iteration, only vertices at distance $\Delta - k$ from v aggregate information from their predecessors, where Δ is the depth of the graph:

$$a^{(k)}_{\{i:\mathcal{D}(i,v)=\Delta-k\}} = \text{AGGREGATE}(h^{(k-1)}u : u \in N(i) \tag{1}$$

The aggregated embeddings are then combined:

$$h^{(k)}_{\{i:\mathcal{D}(i,v)=\Delta-k\}} = \text{COMBINE}(a^{(k)}_i, h^{(0)}_i) \tag{2}$$

In this way, each edge is activated exactly once during the embedding of v, which saves lots of computational efforts compared to the synchronous approach where messages flow on each edge in every iteration. The asynchronous scheme is able to efficiently leverage the acyclic property of DAGs for representation learning.

ACCNN is implemented with asynchronous information propagation in the same way as described. Fig. 3 illustrates the sampled cascaded cones to be embedded by asynchronous message passing. The following information is used as the input feature of *ACCNN*:

- Node type: `Primary Input, Primary Output, CONSTANT, LUT 0x1, LUT 0x2, LUT 0x8.`

For dual input gates (i.e. AND) with only a single input in AIG, we add a constant node as the other input signal, which we believe improves smoothness of message aggregation.

D. SeqEncoder: Optimization Sequence Feature Extraction

In logic synthesis, a sequence of optimization passes offered by the synthesizer are applied to the design netlist, and then are orchestrated to obtain the desired targets including area, timing, etc. Due to the large design space and complex effects between netlists and various optimization passes, designing an optimal optimization sequence is difficult and always undergoes trial-and-errors. Therefore, we take the optimization sequence into consideration for a robust timing prediction.

There is a mini example (adopted from [31]) to demonstrate that different optimizations can produce various design options, where the area and delay of the design are considered. As Fig. 4 illustrated, the tree-height reduction is a similar technique that has originally been proposed in the context of (compiler) code generation for multiprocessor systems, where the tree in Fig. 4(a) represents a decomposed form of the Boolean expression $ab + acd + acef + acegh$. Denoting the area and delay value as $\mathcal{A}$ and $\mathcal{D}$, we assume zero arrival time for all primary inputs, unit area ($\mathcal{A} = 1$), and unit delay ($\mathcal{D} = 1$) of each node, and the tree can be characterized as a pair ($\mathcal{A} = 7, \mathcal{D} = 7$). As implementation from SIS optimization [32], simple transformation by associative and distributive laws, and further optimizations are illustrated in Fig. 4(b), Fig. 4(c), and Fig. 4(d), respectively. It can be verified that they are all functionally equivalent, but with various characteristics: ($\mathcal{A} = 9, \mathcal{D} = 5$) for (b), ($\mathcal{A} = 8, \mathcal{D} = 5$) for (c), and ($\mathcal{A} = 9, \mathcal{D} = 4$) for (d).

As shown in Fig. 4(e), the implementation in (a) is area-optimal and the implementation in (d) is delay-optimal for this case. As for design space exploration, the design in (b) can be discarded, since it is dominant by the design in (c), which achieves better area under the same delay. Generally, even for the tiny example, it is difficult to know beforehand how to obtain the optimal design. While in reality, the designs have larger magnitudes and involve more complex topology and functional structures. Therefore, it is hardly possible for designers to determine the effect of optimization sequences for different designs.

To solve this issue, it is appropriate for LSTP to employ machine learning techniques, which has been served as a suitable black-box predictor for such fuzzy cases. LSTP is then capable to mine its hidden relationships by feeding optimization sequences into the neural network to predict the results in a highly efficient manner. In other words, we need a model that takes into account optimization sequence ordering and position. Many natural language processing models have been designed to learn the order, position, and relationships of words, making them good candidates for our task.

Transformer [40] is one of such models, whose encoder employs positional encoding and multi-head self-attention. The positional encoding is added to the input embedding in the encoder, which provides the positional information of the sequence. As a variant of attention mechanism, *Self-attention* can effectively deal with long-range dependencies and allows for a better understanding of the structural information of the

TABLE I Statistics of the dataset for timing prediction.

Type	IP	
	Train	Valid/Test
Bus protocol	i2c [33], spi [33] ethernet [33], wb_dma [33] simple_spi [33], pci [33] wb_conmax [33]	usb_phy [33] ss_pcm [33] sasc [33]
Controller	ac97_ctrl [33], bp_be [34] vga_lcd [33]	mem_ctrl [33]
Crypto	aes_secworks [35] aes_xcrypt [36] sha256 [37]	aes [33] des3_area [33]
DSP	fir [37], iir [37], jpeg [33]	dft [37], idft [37]
Processor	dynamic_node [38] picosoc [39], tv80 [33]	fpu [39], tinyRocket [38]

TABLE II Evaluation Accuracy (MAPE)

Name	# PI	# PO	# Node	# Level	SNS [4]	Rt [s]	LSTP	Rt [s]
aes	683	529	39215	44	50.21%	2.85	**25.44%**	3.38
des3_area	303	64	7766	47	53.84%	2.16	**20.29%**	0.70
dft	37597	37417	488165	83	86.90%	27.18	**33.56%**	55.53
fpu	632	409	55935	1522	26.11%	4.96	**3.35%**	6.97
idft	37603	37419	481184	82	**5.07%**	16.16	8.18%	54.04
mem_ctrl	1187	962	29814	56	23.21%	32.02	**19.22%**	3.71
sasc	135	125	1214	15	21.44%	2.38	**2.48%**	0.12
ss_pcm	104	90	762	13	67.82%	2.30	**6.46%**	0.09
tinyRocket	4561	4181	99775	156	37.31%	81.87	**10.81%**	11.86
usb_phy	132	90	893	16	14.4%	2.65	**7.75%**	0.10
Average					38.63%	17.45	**13.75%**	**13.65**

sentence, which is computed as:

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d_k}})\boldsymbol{V}, \quad (3)$$

where $(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V})$ are transformed by the multiplication of the input data $\boldsymbol{X}$ and the training parameters $(\boldsymbol{W}_Q, \boldsymbol{W}_K, \boldsymbol{W}_V)$, and $\sqrt{d_k}$ is defined as the feature dimension to alleviate the problem of gradient disappearance. By using positional encoding and multi-head self-attention, *Transformer* is highly parallelized and leads to superior performance in language modeling.

We designed a network named *SeqEncoder* based on Transformer encoder to extract optimization sequence features. Currently, LSTP supports optimization methods such as `Balancing, Reconfiguration, Replacing and Rewriting.`

SeqEncoder supports extracting features of optimization sequences of length 20 or less. When the length of the optimization sequence is less than 20, we need to add padding with zeros before the optimization sequence until the length of the optimization sequence is 20 (i.e., zero padding). To keep the semantics consistent, the dictionary index 0 is considered as '*empty optimization*' that exactly performs nothing to the netlist. During the encoding process, each step of the optimization performed is equivalent to generating a new design, whose remaining steps in the sequence can also be considered as a new optimization sequence operating on the newly generated design. Therefore, the dictionary design can help LSTP to learn a large amount of information with a few designs. Thus, it allows both *ACCNN* and *SeqEncoder* to process a good generalization capability in the training process.

IV. Experiments

We conducted comprehensive experiments to evaluate the performance of LSTP . All experiments are carried out on a machine with 8 core Intel(R) Core(TM) i7-11700 @ 2.50GHz and an NVIDIA GeForce GTX 1660 Ti (Turing architecture, SM67) graphics card with CUDA Driver 11.7, PyTorch 1.13.1, and PyTorch Geometric 2.3.1.

We use SNS [4] as the comparison baseline as it shares a similar scope with ours. The performance metrics include mean absolute percentage error (MAPE) between the predicted timing and the actual timing. We will also show that using LSTP can help find a better optimization sequence.

A. Dataset Generation

We employed Yosys and ABC in tandem for the logic synthesis of the circuit. Yosys, serving as the front-end engine, performed logic synthesis by taking the circuit's source code as input. Sequential logic optimization was subsequently carried out, while the combinational part of the circuit was handed over to ABC. Utilizing structured processing techniques, ABC created an AIG in the BENCH file format.

To generate timing labels for our dataset, we utilized ABC for technology mapping of the netlist. The resulting technologically mapped netlist was then imported into commercial tools for getting the timing characteristics of the circuit.

We selected the circuit listed in Table TABLE I as the benchmark circuit for our experiment. These designs are open-source and also included in the OpenABC-D [19] dataset. Each circuit is synthesized with 1500 optimization sequences, and each optimization sequence is of length $L = 20$. As introduced in Section III-D, these optimization sequences consist of seven structural transformations including `refactor`, `refactor -z`, `rewrite`, `rewrite -z`, `resub`, `resub -z`, and `balance`. The total size of the dataset is 43500, and the labels are obtained after technology mapping with NanGate 45nm technology library and the "5K_heavy_1k" wireload model.

B. Performance of LSTP

We compare the prediction accuracy between LSTP and SNS, and the results are shown in TABLE II. Compared with SNS, the average prediction accuracy (measured by MAPE) of LSTP is improved by 24.88%, and for 9 out of 10 benchmark designs in the validation dataset, LSTP outperforms SNS. Only for one designs (*idft*), the prediction of SNS is more accurate. LSTP wins all other cases.

C. LSTP for Optimization Sequence Generation

We evaluate LSTP with the optimization sequences in the test set as illustrated in TABLE I, to predict the timing of each design. To ensure fairness in comparison, we conducted two rounds of *resyn2* sequences (i.e., invoking the optimization method 20 times, with each *resyn2* calling the optimization method 10 times). We refer to this as *resyn2-2*. For each design, we provided LSTP with 1500 random optimization sequences

TABLE III Comparison of Timing minimums.

Name	Initial [ns]	*r2* [ns]	Impr [%]	LSTP [ns]	Impr [%]
aes	1.58	1.37	13.29%	1.24	**21.52%**
des3_area	2.66	3.74	-40.60%	3.33	-25.19%
dft	5.82	6.35	-9.11%	4.94	**15.12%**
fpu	51	41.5	18.63%	40.34	**20.90%**
idft	5.82	6.35	-9.11%	5.54	**4.81%**
mem_ctrl	6.74	3.03	55.04%	2.94	**56.38%**
sasc	0.89	0.69	22.47%	0.49	**44.94%**
ss_pcm	0.66	0.58	12.12%	0.48	**27.27%**
tinyRocket	78.1	12	84.64%	10.39	**86.70%**
usb_phy	0.41	0.42	-2.44%	0.32	**21.95%**
Average			14.49%		**27.44%**

of length 20. Subsequently, we selected the optimization sequences that exhibited the TOP 10 timing predictions and evaluated their actual execution time to assess the effectiveness of the approach.

As the results shown in TABLE III, the timing error is noticeably low in most designs, which indicates that LSTP can find better optimization sequences. Compared with SNS [4] that predicts timing assuming default tool settings, LSTP takes logic synthesis optimization sequences into consideration during timing prediction, which generates more accurate and case-dependent timing information.

V. Conclusion

Various tasks from architectural exploration to physical design DSE have highlighted the demand for fast logic synthesis result prediction. To this end, we have proposed a machine learning driven logic synthesis timing predictor, LSTP, that efficiently predicts post-synthesis timing of arbitrary circuit design. LSTP models the complex interaction between optimization passes and their effects on the input netlist via appropriate neural models, which is the key to its success. Comprehensive experiments on real-world circuit designs have shown its superior prediction accuracy compared with state-of-the-art timing predictors. Moreover, LSTP has demonstrated its ability to guide optimization sequence generation, which indeed helps to improve design performance. As for the future work, the proposed framework can be extended to predict other physical characteristics of interests, such as area and power.

VI. Acknowledgement

This work is supported by Shanghai Artiticial Intelligence Laboratory (No. P22KN00111).

References

[1] L. Lavagno, L. Scheffer, and G. Martin, *EDA for IC implementation, circuit design, and process technology*. CRC press, 2018.
[2] E. Testa, M. Soeken, L. G. Amar, and G. De Micheli, "Logic synthesis for established and emerging computing," *Proceedings of the IEEE*, 2018.
[3] C. Bai, Q. Sun, J. Zhai, Y. Ma, B. Yu, and M. D. Wong, "Boom-explorer: Risc-v boom microarchitecture design space exploration framework," in *Proc. ICCAD*, 2021.
[4] C. Xu, C. Kjellqvist, and L. W. Wills, "Sns's not a synthesizer: A deep-learning-based synthesis predictor," in *Proc. ISCA*, 2022.
[5] A. B. Kahng, "Machine learning applications in physical design: Recent results and directions," in *Proc. ISPD*, 2018.
[6] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic minimization algorithms for VLSI synthesis*. Springer Science & Business Media, 1984.
[7] G. D. Micheli, *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.
[8] A. Mishchenko and R. K. Brayton, "Sat-based complete don't-care computation for network optimization," in *Proc. DATE*, 2005.
[9] N. Sorensson and N. Een, "Minisat v1. 13-a sat solver with conflict-clause minimization," *SAT*, 2005.
[10] H. Kanakia, M. Nazemi, A. Fayyazi, and M. Pedram, "Espresso-gpu: blazingly fast two-level logic minimization," in *Proc. DATE*, 2021.
[11] G. Pasandi, S. Pratty, D. Brown, Y. Zhang, H. Ren, and B. Khailany, "2021 iccad cad contest problem c: Gpu accelerated logic rewriting," in *Proc. ICCAD*, 2021.
[12] E. Ustun, C. Deng, D. Pal, Z. Li, and Z. Zhang, "Accurate operation delay prediction for FPGA HLS using graph neural networks," in *Proc. ICCAD*, 2020.
[13] C. Yu and W. Zhou, "Decision making in synthesis cross technologies using lstms and transfer learning," in *Proc. MLCAD*, 2020.
[14] Z. Xie, H. Ren, B. Khailany, Y. Sheng, S. Santosh, J. Hu, and Y. Chen, "PowerNet: Transferable dynamic IR drop estimation via maximum convolutional neural network," in *Proc. ASPDAC*, 2020.
[15] Y. Zhang, H. Ren, and B. Khailany, "GRANNITE: Graph neural network inference for transferable power estimation," in *Proc. DAC*, 2020.
[16] S. De, M. Shafique, and H. Corporaal, "Delay prediction for asic hls: Comparing graph-based and non-graph-based learning models," *IEEE TCAD*, 2022.
[17] T. Chen, Q. Sun, C. Zhan, C. Liu, H. Yu, and B. Yu, "Deep h-gcn: Fast analog ic aging-induced degradation estimation," *IEEE TCAD*, 2021.
[18] N. Wu, J. Lee, Y. Xie, and C. Hao, "Lostin: Logic optimization via spatio-temporal information with hybrid graph models," in *Proc. ASAP*, 2022.
[19] A. B. Chowdhury, B. Tan, R. Karri, and S. Garg, "Openabc-d: A large-scale dataset for machine learning guided integrated circuit synthesis," *arXiv preprint arXiv:2110.11292*, 2021.
[20] C. Yu, H. Xiao, and G. De Micheli, "Developing synthesis flows without human knowledge," in *Proc. DAC*, 2018.
[21] W. Haaswijk, E. Collins, B. Seguin, M. Soeken, F. Kaplan, S. Süsstrunk, and G. De Micheli, "Deep learning for logic optimization algorithms," in *Proc. ISCAS*, 2018.
[22] K. Zhu, M. Liu, H. Chen, Z. Zhao, and D. Z. Pan, "Exploring logic optimizations with reinforcement learning and graph convolutional network," in *Proc. MLCAD*, 2020.
[23] R. Brummayer and A. Biere, "Local two-level and-inverter graph minimization without blowup," *Proc. MEMICS*, 2006.
[24] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *Proc. CAV*, 2010.
[25] C. Wolf, "Yosys open synthesis suite," 2016.
[26] Z. He, Z. Wang, C. Bail, H. Yang, and B. Yu, "Graph learning-based arithmetic block identification," in *Proc. ICCAD*, 2021.
[27] L. Alrahis, A. Sengupta, J. Knechtel, S. Patnaik, H. Saleh, B. Mohammad, M. Al-Qutayri, and O. Sinanoglu, "Gnn-re: Graph neural networks for reverse engineering of gate-level netlists," *IEEE TCAD*, 2021.
[28] Z. Wang, C. Bai, Z. He, G. Zhang, Q. Xu, T.-Y. Ho, B. Yu, and Y. Huang, "Functionality matters in netlist representation learning," in *Proc. DAC*, 2022.
[29] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graph-SAINT: Graph sampling based inductive learning method," in *Proc. ICLR*, 2020.
[30] K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Communications of the ACM*, 1981.
[31] J. Cortadella, "Timing-driven logic bi-decomposition," *IEEE TCAD*, 2003.
[32] K. J. Singh, A. R. Wang, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Timing optimization of combinational logic." in *Proc. ICCAD*, 1988.
[33] "Opencores hardware rtl designs," https://opencores.org.
[34] "Black parrot soc," https://github.com/black-parrot/black-parrot.
[35] "Aes 128/256-bit symmetric block cipher," https://github.com/secworks/aes.
[36] "Aes 128/256-bit symmetric block cipher," https://github.com/crypt-xie/XCryptCore/tree/master/ciphers/aes.
[37] "Mit common evaluation platform(cep)," https://github.com/mit-ll/CEP.
[38] T. Ajayi and D. Blaauw, "Openroad: Toward a self-driving, open-source digital layout implementation tool chain," in *Proc. GOMACTech*, 2019.
[39] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang *et al.*, "Openpiton: An open source manycore research framework," *ACM SIGPLAN Notices*, 2016.
[40] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," 2017.