HLS-Timer: Fine-Grained Path-Level Timing Estimation for High-Level Synthesis

Zibo Hu[1], Zhe Lin[2†], Renjing Hou[1], Xingyu Qin[1], Jianwang Zhai[1], and Kang Zhao[1†],
[1]Beijing University of Posts and Telecommunications, Beijing [2]Sun Yat-sen University, Shenzhen Campus
{huzibo, hourenjing, qinxingyu, zhaijw, zhaokang}@bupt.edu.cn, linzh235@mail.sysu.edu.cn

***Abstract*—Electronic Design Automation (EDA) requires early stage timing guidance to maximize optimization potential. Accurate timing estimation is essential in the High-Level Synthesis (HLS) stage. However, current HLS tools often produce inaccurate timing predictions, resulting in unmet performance targets. While post-synthesis EDA tool chains can provide precise timing analysis, their exhaustive methodologies are prohibitively time-consuming and computationally expensive. Recent machine learning approaches have demonstrated promising results in predicting design-level timing metrics in HLS designs, such as Worst Negative Slack (WNS) and Critical Path (CP) delay. Nevertheless, fine-grained, path-level timing estimation remains an unresolved challenge. In this work, we present HLS-Timer, the first path-level timing estimator for HLS. The proposed framework employs a graph-based representation of the HLS design, integrating local structural features with global contextual information to model timing paths and provide accurate, fine-grained delay predictions. Experimental results demonstrate that on previously unseen designs HLS-Timer achieves exceptional accuracy in path-level delay estimation (Pearson R = 0.94, R^2 = 0.93, MAPE = 18.96%), highlighting its strong generalization capability. Furthermore, it surpasses state-of-the-art baselines in design-level timing prediction, reducing MAPE to 9.97% for WNS and 6.79% for CP delays.**

I. Introduction

High-Level Synthesis (HLS) is an automated process that translates hardware designs written in high-level programming languages (e.g., C/C++) into Register Transfer Level (RTL) representations. By enabling designers to work at a high level of abstraction, HLS significantly reduces the complexity of hardware design, shortens development time, and accelerates design iteration. Scheduling is a critical phase in the HLS back-end flow. It assigns computational operations to specific clock cycles during execution, relying on accurate timing to resolve temporal conflicts. Traditionally, this process requires precise, cycle-accurate timing information to model dynamic dependencies and resource contention effectively. However, current HLS tools lack access to accurate timing data. They often rely on fixed latency estimates for each operator type, which overlook key physical factors such as interconnect delays, layout-related effects, and process variations. As a result, the timing guidance used during scheduling is frequently inaccurate, leading to suboptimal scheduling decisions and designs that may not achieve targets.

Conventionally, obtaining and leveraging accurate timing characteristics for optimization requires traversing the entire Electronic Design Automation (EDA) tool chains. This process imposes a prohibitive computational cost while producing timing metrics that often lack architectural correspondence with HLS-level abstractions. This fundamental mismatch forces HLS tools into a trade-off: either 1) endure iterative physical implementation cycles that undermine HLS productivity benefits, or 2) rely on overly simplified timing assumptions that compromise implementation quality.

Early-stage timing estimation in HLS has been a significant research area. Initial efforts [1], [2] focused on model-based frameworks and machine learning (ML) for Quality-of-Results (QoR) estimation, bridging HLS abstractions with post-implementation metrics. While the Pyramid framework [3] advanced joint timing and resource prediction, its coarse-grained models lacked path-level detail. More recent work [4], [5], has aimed for greater granularity and accuracy, with Graph Neural Network (GNN)-based approaches predicting critical path (CP) delays directly from HLS designs. TABLE I summarizes ML-based approaches in HLS-stage timing prediction. However, these methods often fail to establish explicit mappings between HLS primitives and physical-design Static Timing Analysis (STA) metrics, such as path-level slack.

RTL timing estimation remains challenging due to absent physical implementation details. Early works [6] used coarse metrics for PPA. More recent studies [7]–[9] applied GNNs for path-level delays and critical component identification, yet generalizability and actionable guidance persist as open challenges. At the Netlist and Layout-Level stages, GNNs [10], [11] and Transformers [12], [13] have advanced prediction, but their reliance on post-synthesis or post-layout data limits their utility in early design space exploration.

We summarize two major challenges in achieving timing prediction at the HLS stage. 1) There is no direct mapping between HLS designs and gate-level netlists, where timing characteristics can be obtained from STA reports.2) Timing characteristics are complex, influenced by physical implementation, process variations, and architectural decisions, making accurate path-level timing prediction difficult in early design.

In this work, we propose HLS-Timer, a novel framework designed to address the aforementioned challenges. The overall architecture of HLS-Timer is illustrated in Fig. 1. 1) We model the HLS design as a graph and perform timing path sampling guided by data flow, using the reverse-annotated STA timing information as labels for the sampled paths. 2) To address challenge 1, we observe a structural correspondence between HLS intermediate representation (IR) nodes and RTL components, and further identify a consistency [15] between registers in RTL and those in the gate-level netlist. Based on this, we introduce a two-stage mapping strategy that enables

†Corresponding author.

TABLE I Comparison of ML-based approaches for High-Level Synthesis (HLS) Timing prediction tasks.

Work	ML model		Target		Task	Feature Source	Tool
	Graph	Non-Graph	FPGA	ASIC			
Pyramid [3]		✓	✓		Resource Usage and Timing	HLS reports	Vivado HLS
Wu et al. [4]	✓		✓		Resource Usage and CP Timing	IR operator information and HLS reports	Vitis HLS
Dai et al. [2]		✓	✓		Resource Usage and Timing	HLS reports	Vivado HLS
De et al. [14]	✓	✓		✓	Timing	HLS reports	Stratus HLS
This Work	✓	✓	✓		**Data delay**, WNS and CP Timing	IR operator information and bind result	Vitis HLS

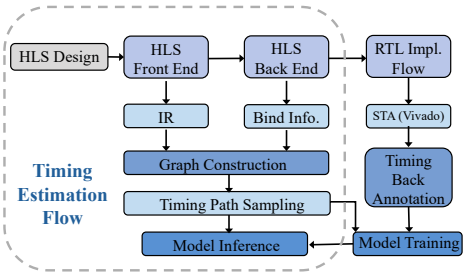


Fig. 1 Overview of the HLS-Timer framework.

back-annotation of timing path information from post-netlist STA reports back to the HLS level. 3) To address challenge 2, we train an end-to-end neural network that aggregates both local and global structural features from the HLS graph to perform fine-grained path-level delay prediction. Furthermore, we leverage the path-level prediction results to estimate design-level metrics, including worst negative slack (WNS) and CP Timing.

Our main contributions are summarized as follows:

- We develop a structured graph representation for HLS designs, explicitly modeling timing endpoints and paths through four key strategies.
- We propose a dataflow-guided path sampling method with back-annotation to propagate gate-level timing back to HLS.
- Our end-to-end model captures both local and global dependencies in HLS IR graphs for fine-grained delay prediction.
- To our knowledge, we present the first framework, HLS-Timer, for fine-grained path-level timing prediction in HLS designs. Our model demonstrates strong performance, achieving an R^2 of 0.93 for fine-grained prediction. For design-level metrics, HLS-Timer outperforms existing state-of-the-art approaches, with MAPE values of 9.97% for WNS and 6.79% for CP.

II. Preliminaries

A. Basic Concepts

Our objective is to enable fast and accurate fine-grained path-level timing estimation at the HLS stage. To clearly define the scope of problems, we introduce the following key concepts:

Definition 1 (Timing Path). A timing path is the signal propagation route that starts at the clock edge of a source endpoint and ends at the clock edge of a destination endpoint. Common types of timing paths include: 1) Register-to-register. 2) Input port-to-register. 3) Register-to-output port. 4) Input port-to-output port.

Definition 2 (Endpoint). An endpoint refers to the start or end of a timing path, typically corresponding to a register or an input/output (I/O) port.

Definition 3 (Data Delay). Data delay is the total time required for a signal to travel from the source endpoint to the destination endpoint along a timing path. It consists of two main components: gate delay and wire delay.

Definition 4 (Static Timing Analysis). STA is a widely used technique in EDA for verifying whether a digital circuit meets its timing constraints without requiring input stimuli. It analyzes all possible signal paths in a design by statically propagating delays through the gate-level netlist, based on pre-characterized timing models under fixed process, voltage, and temperature (PVT) conditions. STA calculates the arrival time (AT) and compares it with the required arrival time (RAT) to compute the slack, which is used to identify timing violations and determine critical paths.

B. Description of Task

We represent the HLS design as H, the synthesized HDL code as V, and the logic-synthesized gate-level netlist as N. Each timing path sampled from the HLS, denoted as p_i, is assigned a ground truth timing value on the gate-level netlist N after performing STA using our proposed two-stage mapping method. Our proposed model M predicts the corresponding ground truth timing value. We formulate our timing estimation problem as follows.

Problem 1. (Data delay of timing path estimation)

For all $p_i \in H$, the model M predicts the data delay $Dt_N(p_i)$ for each timing path,which corresponds to the ground truth timing value in the gate-level netlist N:

$$\forall p_i \in H, M_{Dt}(p_i) \rightarrow Dt_{\mathrm{N}}(p_i) \tag{1}$$

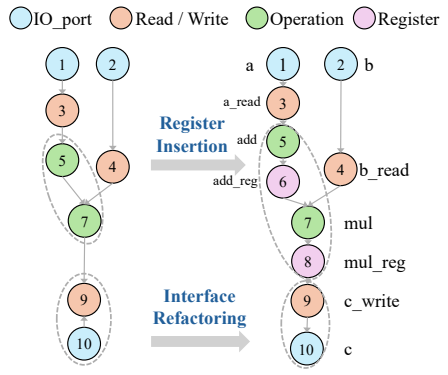
Problem 2. (CP and WNS estimation)

After obtaining the fine-grained prediction results from the Problem 1, we use these as input to predict the global metrics, CP and WNS. The predicted values of the timing paths data delay is denoted as Dt.

$$M_{CP}(Dt) \rightarrow \mathrm{CP} \tag{2}$$
$$M_{WNS}(Dt) \rightarrow \mathrm{WNS} \tag{3}$$

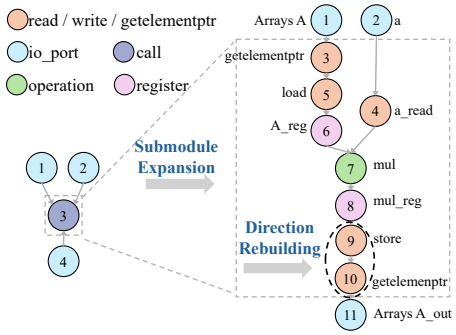
III. Methodology

Our proposed approach HLS-Timer, is outlined in Fig. 1. It seamlessly integrates into Vitis HLS and Vivado workflows, delivering accurate, fine-grained timing predictions at HLS stage. These predictions serve a dual purpose: aidding HLS design optimization and guiding design space



(a) Original IR Graph (b) Constructed Graph

Fig. 2 Register Insertion and Interface Refactoring.



(a) Original IR Graph (b) Constructed Graph

Fig. 3 Submodule Expansion and Direction Rebuilding.

exploration, while also enabling targeted timing optimization during physical implementation. The following subsections detail the graph construction, timing path sampling and back-annotation, and our hierarchical prediction model.

A. Graph Construction

We have designed a graph construction flow to generate graph-structured representations of HLS designs, crucial for efficiently extracting timing paths for downstream tasks like sampling and ground-truth annotation. While HLS compilation provides an initial IR, it is not directly suitable for timing prediction due to its lack of explicit timing endpoint representations. To overcome this, we have introduced four enhancement strategies(shown in Figs. 2 and 3) that systematically augment the HLS IR graph, allowing us to explicitly represent these endpoints and effectively sample timing paths. Each of these four strategies operates with low coupling between different operations, enabling parallel execution to reduce runtime. Consequently, the graph construction runtime does not significantly increase with larger design scales.

Register Insertion. The initial IR poses a challenge: registers, essential as timing path endpoints, are implicitly merged with their upstream dataflow operations. This lack of explicit representation impedes accurate timing path identification. To resolve this, we propose a method for inserting register nodes into the graph representation. Our approach leverages scheduling and binding information from HLS to establish a variable-to-register mapping. Specifically, post-HLS binding, we extract correspondences between IR-level variables and their associated RTL registers. Guided by these mappings, we insert register nodes into the IR graph—typically immediately following their driving operation nodes(e.g., mul, add). As illustrated in Fig. 2, this ensures explicit timing path endpoints (e.g., register nodes 6 and 8 inserted after operation nodes 5 and 7, respectively), thereby enabling precise path sampling.

Interface Refactoring. Beyond registers, I/O ports also serve as critical endpoints of timing paths. However, the initial HLS IR presents two significant limitations that impede accurate timing path extraction. 1) Edge directions for I/O ports are often incorrect, disrupting proper connectivity in the directed graph. 2) The IR graph lacks essential interface semantics; the transformation of C-level function parameters into RTL hardware ports during interface synthesis (e.g., single to multiple, or to bidirectional ports) is not reflected. This omission means many valid timing path endpoints are missing. To address these issues, we revise the IR graph by correcting I/O port edge directions and adjusting port quantity and types based on interface synthesis mappings. This refactoring ensures the graph accurately mirrors the hardware interface, enabling more precise and comprehensive timing path sampling.

Submodule Expansion. The original IR's hierarchical nodes, which abstract submodules into single call operations, significantly limit timing analysis granularity and obscure underlying structural and temporal characteristics. To overcome this, we propose an expansion strategy that replaces each hierarchical call node with the corresponding Control Data Flow Graphs (CDFGs) of the submodule. This approach ensures consistent sampling granularity across all derived graphs and facilitates the accurate extraction of real timing paths, aligning with static timing analysis. While Fig. 3 provides a conceptual illustration, actual implementations involve far greater architectural complexity, with diverse node typologies beyond basic port configurations and call-type operational vertices.

Direction Rebuilding. The init IR inherently contains erroneous edge directions, which compromise graph connectivity and impede the construction of valid data paths from inputs to outputs. For instance, in operations such as getelementptr and store, where getelementptr retrieves an address and store writes data to it, the original IR incorrectly directs the edge from getelementptr to store, despite the actual data flow being in the reverse. A similar misdirection occurs between alloca and store operations. By correcting these edge directions, we ensure accurate graph connectivity that precisely reflects the true data flow, as conceptually illustrated for the store and getelementptr case in Fig. 3.

B. Timing path sampling and back-annotation

We propose a dataflow-driven timing path sampling algorithm to systematically extract timing paths from the graph representation specifically designed for HLS. We further implement a two-phase back-annotation framework to achieve precise temporal label alignment. Timing annotations extracted from gate-level STA are progressively mapped back to HLS-level timing paths through a combination of hierarchical feature mapping and constraint-aware verification, ensuring accurate correspondence across abstraction layers.

Algorithm 1 Timing Path Sampling (Simplified Main Procedure)

Input: Graph G, Initial Start Nodes $S_{initial}$
Output: All Timing Paths P_{all}

procedure `Timing Path Sampling`(G, $S_{initial}$)
  $P_{all} \leftarrow \varnothing$
  $S_{current} \leftarrow S_{initial}$ // List of start nodes for DFS in the current iteration
  $S_{processed\_starts} \leftarrow \varnothing$ // Set of all nodes from which DFS has been initiated
  while $S_{current}$ is not empty **do**
    $P_{round} \leftarrow \varnothing$ // Timing paths newly discovered in this round
    $S_{next\_potential\_starts} \leftarrow \varnothing$ // Set of potential start nodes for the next iteration
    for each node s in $S_{current}$ **do**
      if s is not in $S_{processed\_starts}$ **then**
        `dfs_path_sampling`(G, s, P_{round}) // Call DFS function to explore paths starting from s and add them to P_{round}
        Add s to $S_{processed\_starts}$
        for each path p in P_{round} that started from s **do**
          Add `get_end_node`(p) to $S_{next\_potential\_starts}$
        end for
      end if
    end for
    $P_{all} \leftarrow P_{all} \cup P_{round}$
    $S_{current} \leftarrow S_{next\_potential\_starts} \setminus S_{processed\_starts}$ // Update $S_{current}$ with new, unprocessed start nodes
  end while
  return P_{all}
end procedure

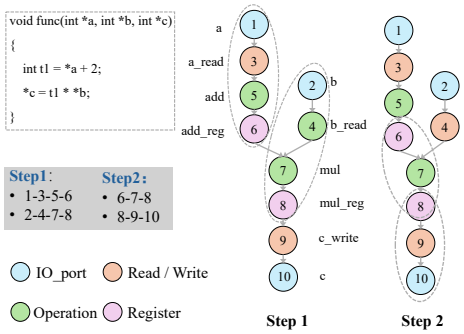


Fig. 4 Example of Timing Path Sampling.

Timing Path Sampling. To obtain timing paths at the HLS level for data delay prediction, HLS-Timer employs a recursive path sampling strategy illustrated in Algorithm 1. The sampling process begins with io port nodes that have an in-degree of 0 and recursively traverses the dataflow graph to capture all timing paths. The main procedure iteratively expands the sampling frontier by performing a depth-first traversal from each unvisited start node. After each round, the set of start nodes is updated based on newly discovered endpoints, and sampling continues until no new nodes are found. An example of the timing path sampling process is illustrated in Fig. 4. Initially, nodes 1 and 2 are selected as start nodes, and the first round of DFS-based sampling generates two timing paths. In the subsequent round, the start nodes are updated to nodes 6 and 8, leading to a second round of sampling. As no additional start nodes emerge after this, the sampling process concludes, resulting in a total of four timing paths.

Netlist to RTL. STA is performed on the gate-level netlist. Given the consistency of register definitions between the RTL and the netlist, timing information from the STA report can be accurately back-annotated onto the RTL. However, a granularity mismatch exists: registers in the netlist are bit-level, whereas in RTL they are typically word-level. Inspired by the timing propagation mechanism in STA, where each endpoint accumulates arrival times from all its driving registers and selects the maximum for slack computation, we apply a similar principle in mapping timing paths onto the RTL. That is, the timing delay associated with an RTL-level path corresponds to the longest (slowest) bit-level timing path in the netlist. Specifically, as long as the start and end points in the netlist belong to the same word-level register in RTL, we annotate the RTL timing path with the maximum data delay observed across all corresponding bit-level paths. This relationship is formally defined as follows:

$$D_{\text{RTL}}(S,T) = \max_{r_i \in R} D_{\text{Netlist}}(s_i, t_i) \tag{4}$$

where S/T denotes a word-level register Start/Target in RTL, and s_i/t_i represents its bit-level components in the gate-level netlist.

IR to RTL. The HLS execution process consists of three main stages: front-end processing, back-end processing, and RTL generation. Following the front-end stage, the HLS design is compiled into the IR, with node names largely consistent with the original C code. The back-end stage then performs scheduling and binding, generating a mapping between IR-level and RTL-level names. After completion of the binding stage, HLS tools provide information that enables the establishment of a bidirectional mapping between IR names and corresponding RTL component names.

C. Hierarchical Estimation Model

The mainstream approaches for timing analysis in HLS predominantly employ Graph Neural Networks (GNNs) for design-level prediction. However, conventional GNN architectures face inherent limitations due to layer depth constraints: insufficient layers result in restricted receptive fields and inadequate expressive power, leading to underfitting, while excessive layers induce node representation homogeneity and training difficulties caused by over-smoothing phenomena. Traditional GNN implementations struggle to effectively integrate both localized structural patterns and global topological characteristics. To address these limitations, we propose an end-to-end neural architecture (As shown in Fig. 5) comprising three principal components: a GNN module, a Transformer module, and a Multilayer Perceptrons (MLP). This integrated framework enables comprehensive feature learning across multiple granularities. The standardized GNN component extracts localized structural information through neighborhood feature aggregation. Subsequently, the Transformer module captures global contextual dependencies via cross-node attention mechanisms. Building upon these complementary representations, the adaptive pooling module synthesizes path-level embeddings through hierarchical feature composition. Finally, the MLP classifier utilizes these path representations to achieve rapid and accurate prediction of data delays for individual timing paths. Furthermore, our methodology establishes a two-stage prediction paradigm: The path-level delay estimations subsequently serve as input features

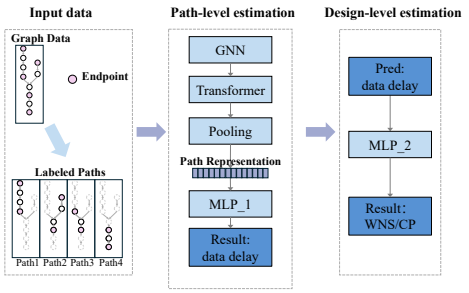


Fig. 5 Hierarchical Estimation Model

for secondary regression models that predict system-level timing metrics, specifically WNS and CP, thereby enabling comprehensive timing analysis. The subsequent sections will provide a comprehensive elaboration of the proposed model architecture, the employed feature set, and the experimental task formulation.

Dataset. Our dataset contains both synthetic C programs and real-world cases. The synthetic portion includes 12,653 HLS-compliant C implementations generated through automated methods. Real-world cases are sourced from MachSuite [16] and PolyBench/C [17], including 39 designs. The detailed information of the dataset is shown in TABLE II, including post-implementation resource utilization (Block RAM (BRAM), Look-Up Tables (LUT), Digital Signal Processors (DSP), and Registers (Reg))—where LUTs often serve as a measure of design size—along with the number of nodes in the constructed IR graph, and the count of timing paths after converting the bit-level gate-level netlist paths to their word-level counterparts.

TABLE II Resource and Path Statistics of HLS Benchmarks

Statistic	BRAM	LUT	DSP	Reg	Num_Node	Num_Path
min	0	0	0	0	4	0
max	364	45018	1024	76301	4113	4344
avg	0.18	3408	10	4425	71	85

Node Feature. The node features used by HLS-Timer are summarized in TABLE III. Features such as bit width, opcode, path start indicator, LCD node flag, and cluster group are extracted from the HLS front-end compilation results. In addition, node types and opcode categories are constructed based on domain-specific knowledge.

Tasks/Labels. We define two types of prediction tasks: fine-grained data delay prediction and design-level WNS/CP prediction.

- Data delay regression task: For each timing path extracted from the graph, the corresponding data delay value reported by STA is used as the supervision label. The model learns to predict the data delay of each individual timing path.
- WNS/CP regression task: At the design level, each graph is labeled with the WNS/CP obtained after physical implementation. The prediction is performed by aggregating the fine-grained data delay values of all timing paths predicted in the previous step.

Model. Our neural architecture integrates three core components: GNNs, Transformer modules, and MLPs. The GNN

TABLE III Node Features and Example Values.

Feature	Description	Values
Node category	General node type	port, operation node, reg, etc.
Bitwidth	Bitwidth of the node	0∼256, misc
Opcode category	Opcode categories	binary_unary, bitwise, memory, etc.
Opcode	Opcode of the node	load, add, mul, store, etc.
Is start of path	Whether the node is the starting node of a path	0, 1, misc
Is LCDNode	Whether the node is LCD Node	0, 1, misc
Cluster group	Cluster number of the node	-1∼256, misc

component extracts local structural patterns through neighbor-node information aggregation, where we realize four variants: 1) Graph Convolutional Network (GCN) [18] for spectral neighborhood aggregation; 2) Graph Attention Network (GAT) [19] with learnable attention weights; 3) Graph Isomorphism Network (GIN) [20] employing injective aggregation; 4) GraphSAGE [21] with fixed-size neighborhood sampling to maintain computational consistency.All GNN variants share identical layer configurations and MLP [22] for performance comparison. Subsequently, the Transformer [23] module processes flattened node embeddings via self-attention mechanisms to capture global dependencies. Finally, temporal path representations are generated through node embedding pooling and fed into the MLP regressor for data delay prediction.

IV. Experimental Result

A. Experimental Setup

The framework, encompassing graph construction, timing path sampling, and back-annotation, is implemented in Python, with the neural network architecture built using PyTorch. Datasets are randomly split into 80% train, 10%validation and 10% test , with real-case benchmarks only used for evaluating model generalization. The ground-truth timing labels are synthesized by Vitis HLS 2022.1 [24], and implemented by Vivado 2022.1 [25]. The model architecture contains a 2-layer GNN and a 2-layer Transformer with 4 attention heads, followed by an MLP (64-64-1). Models are trained using Adam optimizer for 100 epochs.

B. Evaluation Metrics

We employ Pearson correlation coefficient (R), coefficient of determination (R^2), and Mean Absolute Percentage Error (MAPE) to evaluate our model's effectiveness. R assesses the linear correlation between predictions and ground truth, ranging from -1 to 1. R^2, spanning 0 to 1, indicates the proportion of variance in the dependent variable predictable from the independent variable. MAPE quantifies prediction accuracy, with lower values signifying better performance. Higher R and R^2 values combined with lower MAPE values indicate superior regression accuracy.

C. Modeling Performance

Path-level Performance. Our path-level delay regression results, summarized in TABLE IV, demonstrate the effectiveness of the proposed approach. Essentially, using neural

TABLE IV Path-level timing accuracy and ablation study.

Fine-Grained	Method	R	R^2	MAPE (%)
data delay	MLP	0.9599	0.9196	20.8682
	Transfomer	0.9560	0.9115	22.3479
	GCN	0.9560	0.9131	25.4475
	GAT	0.9522	0.9055	26.1010
	GIN	0.9574	0.9148	22.5104
	GraphSage	0.9566	0.9126	22.2962
	This Work	0.9416	**0.9273**	**18.9554**

TABLE V MAPE of graph-level indicators CP/WNS

Metric	Method	MAPE(%)
CP	Wu et al. [4]	7.6805
	This Work	**6.7926**
WNS	This Work	**9.9730**

networks to predict path-level timing from IR graphs is to approximate the set of sophisticated heuristics and mapping rules used by HLS scheduling/binding and logic/physical synthesis during design flow. To effectively capture this intricate set of rules, we introduce a hierarchical model designed to robustly extract features for accurate approximation. In constructing the IR graph, we endeavor to reflect the physical synthesis-level circuit topology as closely as possible within the IR. This dual effort—feature extraction via our hierarchical model and meticulous IR graph construction—has enabled us to achieve state-of-the-art performance in path-level timing prediction.

Design-level Performance. Although our primary focus is on path-level timing prediction for HLS designs, we also evaluated our framework's ability to infer design-level timing indicators. We benchmarked our approach against the state-of-the-art graph learning method by Wu et al. [4], which focuses on graph-level CP timing estimation. Our lightweight architecture demonstrated superior Critical Path (CP) prediction accuracy. As TABLE V illustrates, our method not only achieved higher accuracy but also reduced training time by 50% compared to the baseline's significantly deeper model. Furthermore, our framework supports multi-task learning, enabling simultaneous prediction of WNS metrics.

Ablation Studies. To assess the contribution of individual components within our model's architecture, we conduct ablation studies under consistent experimental settings as detailed in Section IV-A. As TABLE IV illustrates, our model achieved the best R² and MAPE, attributable to the GNN capture of local features and the Transformer's ability to extract global features. Among all GNN variants, GraphSAGE demonstrated superior performance, due to its neighborhood sampling strategy, which is better suited for capturing local features in complex graph structures.

Generalization. To validate our model's practical applicability, we thoroughly evaluated it on 20 real-world designs. These diverse cases, spanning linear algebra, stencil computation, and data streaming, provide a robust testbed for assessing timing behavior in realistic HLS scenarios. As TABLE VI shows, our framework consistently achieves high prediction fidelity across all benchmarks, with minimal deviation from post-synthesis timing analysis. These results confirm the model's robustness and generalizability beyond synthetic data, highlighting its potential for use in practical HLS optimization

TABLE VI Performance on Real-word designs.

design	Path-level Pred		Design-level Pred					
	data delay		CP			WNS		
	MAPE(%)	R	Real(ns)	Pred(ns)	MAPE(%)	Real(ns)	Pred(ns)	MAPE(%)
aA	9.020	1.000	10.235	10.208	0.261	-4.092	-4.302	5.139
aAB	18.036	0.962	11.474	10.210	11.017	-4.091	-4.431	8.307
aAplusbB	16.748	0.988	10.827	10.834	0.062	-4.136	-4.369	5.628
AB_1	13.659	0.978	12.021	10.058	16.302	-4.445	-4.510	1.460
AB_2	11.507	0.969	10.574	10.074	4.725	-4.176	-4.375	4.777
ABplusC	11.944	0.983	11.282	10.044	10.974	-4.274	-4.298	0.560
ABplusCD	15.866	0.956	11.630	9.925	14.657	-3.961	-4.187	5.697
ABx_1	27.288	1.000	11.168	9.923	11.144	-7.195	-5.652	21.441
ABx_2	17.733	0.999	11.336	9.719	14.267	-6.012	-4.455	25.906
AplusB	9.523	1.000	11.237	10.465	6.866	-4.192	-4.217	0.587
bfs	23.833	0.821	11.239	11.131	0.957	-6.683	-4.349	34.901
bicg	7.917	1.000	8.692	8.267	4.888	-5.244	-4.405	15.998
gemm	18.815	0.978	11.470	10.285	10.334	-4.420	-4.438	0.402
k3mm	16.937	0.963	11.783	9.522	19.192	-4.217	-4.375	3.744
mac_1	12.467	0.975	11.116	10.868	2.226	-4.445	-4.418	0.610
mac_2	8.941	0.995	10.820	10.972	1.406	-4.444	-4.443	0.014
mac_3	22.463	0.977	11.257	10.697	4.972	-4.406	-4.383	0.528
mac_4	25.101	0.976	11.239	10.684	4.937	-4.465	-4.488	0.507
mac_5	9.229	0.999	10.300	10.409	1.055	-4.492	-4.490	0.040
syrk	20.162	0.927	10.784	10.341	4.112	-4.263	-4.430	3.928
Avg	**15.859**	**0.972**	**11.024**	**10.232**	**7.218**	**-4.683**	**-4.451**	**7.009**

workflows. Crucially, our framework can directly provide fine-grained timing predictions for any unseen design at the HLS stage, provided it complies with Vitis-HLS requirements, without needing extra operations.

D. Runtime Analysis

HLS-Timer provides fast and accurate fine-grained path-level timing estimation, eliminating the need for logic synthesis, placement, routing, and STA processes. Overall, our method achieves a 10x acceleration compared to traditional flow. It comprises two key parts: 1) Data processing: This involves graph construction and timing path sampling. 2) Model inference: This takes less than 0.1 seconds. Specifically, on our dataset, the data processing typically requires 55.38 seconds on average, while Vivado execution averages 586.82 seconds.

V. Conclusion

This paper introduces HLS-Timer, the first fine-grained path-level timing prediction framework for HLS designs. HLS-Timer accurately predicts timing path data delays by leveraging local and global information through novel graph construction and back-annotation. Experimental results show its high accuracy, efficiency, and portability, positioning it as a promising tool for HLS timing optimization. Future work will focus on enhancing prediction accuracy and using these capabilities to optimize HLS scheduling algorithms.

VI. Acknowledgment

This work was supported in part by National Key R&D Program of China (2022YFB2901100), the National Natural Science Foundation of China (No. 62404257), and Shenzhen Science and Technology Program (Grant No. RCBS20231211090600003).

REFERENCES

[1] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "Comba: A comprehensive model-based analysis framework for high level synthesis of real applications," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 430–437.

[2] S. Dai, Y. Zhou, H. Zhang, E. Ustun, E. F. Young, and Z. Zhang, "Fast and accurate estimation of quality of results in high-level synthesis with machine learning," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, pp. 129–132.

[3] H. M. Makrani, F. Farahmand, H. Sayadi, S. Bondi, S. M. P. Dinakarrao, H. Homayoun, and S. Rafatirad, "Pyramid: Machine learning framework to estimate the optimal timing and resource usage of a high-level synthesis design," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019, pp. 397–403.

[4] N. Wu, H. Yang, Y. Xie, P. Li, and C. Hao, "High-level synthesis performance prediction using gnns: Benchmarking, modeling, and advancing," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 49–54.

[5] P. Goswami and D. Bhatia, "Predicting post-route quality of results estimates for hls designs using machine learning," in *2022 23rd International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2022, pp. 45–50.

[6] W. Fang, Y. Lu, S. Liu, Q. Zhang, C. Xu, L. W. Wills, H. Zhang, and Z. Xie, "Masterrtl: A pre-synthesis ppa estimation framework for any rtl design," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.

[7] D. S. Lopera and W. Ecker, "Applying gnns to timing estimation at rtl," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022, pp. 1–8.

[8] D. S. Lopera, I. Subedi, and W. Ecker, "Using graph neural networks for timing estimations of rtl intermediate representations," in *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*. IEEE, 2023, pp. 1–6.

[9] P. Sengupta, A. Tyagi, Y. Chen, and J. Hu, "Early identification of timing critical rtl components using ml based path delay prediction," in *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*. IEEE, 2023, pp. 1–6.

[10] Z. Xie, R. Liang, X. Xu, J. Hu, C.-C. Chang, J. Pan, and Y. Chen, "Preplacement net length and timing estimation by customized graph neural network," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 4667–4680, 2022.

[11] E. C. Barboza, N. Shukla, Y. Chen, and J. Hu, "Machine learning-based pre-routing timing prediction with reduced pessimism," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[12] P. Cao, G. He, and T. Yang, "Tf-predictor: Transformer-based prerouting path delay prediction framework," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 7, pp. 2227–2237, 2022.

[13] Z. Guo, M. Liu, J. Gu, S. Zhang, D. Z. Pan, and Y. Lin, "A timing engine inspired graph neural network model for pre-routing slack prediction," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 1207–1212.

[14] S. De, M. Shafique, and H. Corporaal, "Delay prediction for asic hls: Comparing graph-based and nongraph-based learning models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 4, pp. 1133–1146, 2022.

[15] W. Fang, S. Liu, H. Zhang, and Z. Xie, "Annotating slack directly on your verilog: Fine-grained rtl timing evaluation for early optimization," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.

[16] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "Machsuite: Benchmarks for accelerator design and customized architectures," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2014, pp. 110–119.

[17] L.-N. Pouchet., "Polybench: The polyhedral benchmark suite." 2014.

[18] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[19] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[20] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[21] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[22] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[23] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[24] Vitis high-level synthesis user guide (ug1399). [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls

[25] Xilinx vivado. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html