ATLAS: A Self-Supervised and Cross-Stage Netlist Power Model for Fine-Grained Time-Based Layout Power Analysis

Wenkai Li[†], Yao Lu[†], Wenji Fang, Jing Wang, Qijun Zhang, Zhiyao Xie[*]
Hong Kong University of Science and Technology
{wlidm, yludf, wfang838, jwangjw, qzhangcs}@connect.ust.hk, eezhiyao@ust.hk
([†]equal contribution, [*]corresponding author)

***Abstract*—Accurate power prediction in VLSI design is crucial for effective power optimization, especially as designs get transformed from gate-level netlist to layout stages. However, traditional accurate power simulation requires time-consuming back-end processing and simulation steps, which significantly impede design optimization. To address this, we propose ATLAS, which can predict the ultimate time-based layout power for any new design in the gate-level netlist. To the best of our knowledge, ATLAS is the first work that supports both time-based power simulation and general cross-design power modeling. It achieves such general time-based power modeling by proposing a new pre-training and fine-tuning paradigm customized for circuit power. Targeting golden per-cycle layout power from commercial tools, our ATLAS achieves the mean absolute percentage error (MAPE) of only 0.58%, 0.45%, and 5.12% for the clock tree, register, and combinational power groups, respectively, without any layout information. Overall, the MAPE for the total power of the entire design is $<1\%$, and the inference speed of a workload is significantly faster than the standard flow of commercial tools.**

I. Introduction

Power is an increasingly important objective in modern chip design. As design complexity keeps scaling up, it is increasingly costly for chip designers to get accurate power values of their design. It is even more challenging to simulate time-based (e.g., per-cycle) power values, which enables the analysis of peak power and power fluctuations (Ldi/dt). As Fig. 1 shows, starting at a gate-level netlist, designers need to complete the design whole layout process and then employ target workloads (in .fsdb or .vcd format) to simulate time-based power consumption based on commercial EDA tools [1], [2]. Both layout and power simulation can take days or even weeks for large designs or workloads. In summary, accurate, efficient, time-based power models are in high demand.

Prior Power Modeling Works. In recent years, various novel data-driven power modeling techniques [3]–[9] have been proposed, providing unprecedented early-stage and fast power evaluations based on machine learning (ML) power models. Representative power modeling works have been summarized in Table I. However, no prior works can provide *time-based power* and generalize across *different designs* simultaneously. We categorize existing power modeling solutions into two main types:

- The first type of works (e.g., PRIMAL [3], APOLLO [4]) can provide accurate and time-based power values, but they do not support general cross-design models. Instead, they [3], [4] require training a new design-specific power model from scratch for each new design. This development process, especially the label collection step, is highly time-consuming.
- The second type of works [5]–[9] can provide a general power model that applies to new designs, which are unknown during model training. Perhaps due to the difficulty of generalization, none of these works further provide time-based power values. Moreover, most of these works [5]–[7], [9] do not model power based on different workloads. Instead, they only model the average power value based on the propagation of user-defined toggle rates (i.e., vectorless power values).

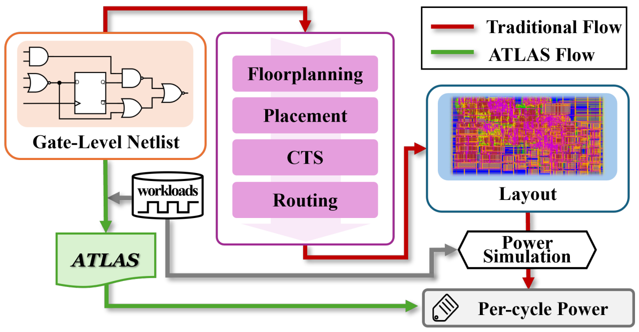


Fig. 1: Overview of ATLAS for time-based netlist power modeling. Standard traditional power simulation for post-layout design is time-consuming due to both layout steps and per-cycle power simulation. ATLAS achieves significant acceleration over the standard flow.

In addition, many prior power modeling solutions [3]–[7] do not model the ultimate power after design layout. For faster data collection, they adopt power values simulated at the gate-level netlist stage as labels, skipping the layout process. As a result, they are not validated on capturing ultimate post-layout power, which is heavily affected by many factors such as accurate metal wire capacitance, timing optimization on netlist (e.g., buffer insertion, netlist reconstruction), the clock tree, etc.

Our Solution ATLAS. In this work, we propose ATLAS, targeting efficient evaluation of the time-based post-layout power of any given gate-level netlist. To the best of our knowledge, ATLAS is the first work that supports both *time-based* power simulation and general *cross-design* power modeling. Moreover, ATLAS targets the most accurate power label from *layout stage* and *realistic designs* (e.g., out-of-order CPU designs instead of small blocks). Compared with standard simulation flow based on EDA tools, the general ATLAS solution bypasses both the layout process and time-based power simulation, achieving significant speedups.

ATLAS achieves unprecedented general time-based power modeling based on a customized *pre-training* and *fine-tuning* paradigm. It proposes the following novel strategies:

- **Sub-module generation:** ATLAS first splits each design into

Power Models	Applied Stage	Support Workloads	Time-Based	Cross-Design	Target Layout
PRIMAL [DAC'20] [3] APOLLO [MICRO'21] [4]	RTL	Yes	Yes	No	No
Sengupta et al. [ICCAD'22] [5] SNS [ISCA'22] [6] SNS V2 [MICRO'23] [7]		No	No	Yes	
MasterRTL [ICCAD'23] [8]		Yes			Yes
PowPredicCT [DAC'24] [9]	Layout	No			
ATLAS	Netlist	**Yes**	**Yes**	**Yes**	**Yes**

[*]GRANNITE [10] estimates toggle rate instead of power, thus not listed in the table. It is neither time-based nor targeting layout.

TABLE I: Summary of representative ML-based power models.

non-overlapping sub-modules. ATLAS will evaluate the per-cycle post-layout power of each small sub-module.

- **Pre-training:** A general netlist encoder model is pre-trained based on multiple self-supervised learning tasks without power labels. The encoder, based on graph transformer, is trained by guessing the masked toggle rate and node type, as well as recognizing the alignment between regular post-synthesis netlist[1] and the ultimate post-layout netlist. The encoder will encode each sub-module into a general embedding (i.e., vector) with rich design information. Such an informative embedding effectively supports challenging power modeling tasks.
- **Fine-tuning:** Based on the embedding from the pre-trained encoder, we fine-tune different lightweight models for three power groups: combinational logic, register, and clock tree.

We evaluate ATLAS on different designs with 300K to 600K cells under realistic workloads. ATLAS achieves high accuracy in per-cycle power modeling, with only $< 1\%$ error percentage on average. ATLAS is up to $1000\times$ faster than the traditional commercial flow by bypassing both the layout process and standard time-based power simulation. The results indicate the superior predictive capability of ATLAS, demonstrating its effectiveness in cross-stage, cross-design, and time-based power prediction.

II. Methodology Overview

This section provides an overview of ATLAS. ATLAS includes three major steps: design netlist preprocessing (Sec. III and Fig. 2), pre-training (Sec. IV and Fig. 3), and fine-tuning (Sec. V and Fig. 4).

Preprocessing (Sec. III). The flow begins with netlist preprocessing, preparing the dataset for ATLAS pre-training. In this step, we split each design in the gate-level netlist into many non-overlapping sub-modules. We transform each sub-module into the graph format and annotate related features (e.g., cell type, cell power from liberty file) to each node (i.e., cell instance). During training, the pre-processed data will be adopted to pre-train the encoder. During inference, ATLAS will encode each sub-module and predict its per-cycle power value.

Pre-training (Sec. IV). The pre-training step will train a general circuit encoder, targeting two general learning goals: 1) recognizing the structure and functionality of netlists; 2) learning the alignment between netlist and layout. This encoder is pre-trained with five novel and customized self-supervised tasks to recognize the inherent semantics and structures of sub-modules, without relying on any power labels. After the pre-training, the encoder is expected to capture the transformations made on each sub-module (e.g., buffer insertion, reconstruction, clock tree) during layout and encode the information into its output embedding vector.

Unlike previous self-supervised learning tasks on netlists [11]–[13] that only target combinational circuits, we target realistic sequential designs. Moreover, we incorporate per-cycle toggle information into one of the five self-supervised learning tasks, allowing ATLAS to learn about signal propagation.

Fine-tuning (Sec. V). The final step of ATLAS fine-tunes different lightweight models for three different power groups: combinational logic, sequential logic, and clock network. These models leverage the embedding vectors generated during pre-training as input, generating fine-grained per-cycle power value of each power group for each sub-module. Summing up all power groups of all sub-modules provides the total power for each cycle.

[1]In this manuscript, if not explicitly described, by default "netlist" refers to the post-synthesis gate-level netlist before layout. The post-layout netlist (with timing optimization and clock tree) will be explicitly mentioned.

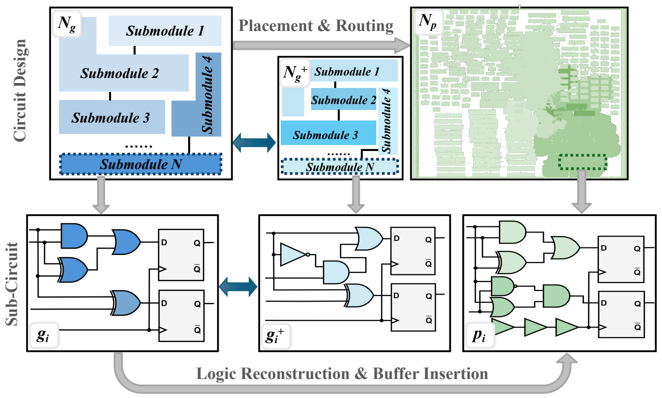


Fig. 2: Netlist Preprocessing of ATLAS. The preprocess includes circuit segmentation in both gate-level netlist N_g, N_g^+ and post-layout netlist N_p. N_g^+ represents a Boolean-equivalent expression of N_g and both are functionally equivalent with similar circuit structures. For each sub-circuit g_i in N_g, there exists an equivalent gate-level expression g_i^+ and a transformed expression p_i after layout.

III. Netlist Preprocessing

The netlist preprocessing step is essential for preparing the dataset used in ATLAS. This involves generating netlists from different stages and collecting key features that facilitate effective learning.

A. Sub-module Format

Fig. 2 illustrates the netlist preprocessing process. Given a gate-level netlist N_g, we split it into a set of non-overlapping sub-modules $N_g \in \{g_1, g_2, ..., g_i\}$. Then this original netlist N_g will be transformed to a corresponding post-layout netlist N_p, which will similarly yield a set of sub-modules $N_p \in \{p_1, p_2, ..., p_i\}$. During this standard layout process, the netlist will be optimized for better timing through buffer insertion, netlist reconstruction, etc. Also, the clock tree will be synthesized and added to the design.

Most previous works on circuit quality prediction [6]–[8] choose to split circuits into multiple *logic cones*, then model the PPA of each logic cone. Each input code corresponds to one flip-flop, including all input logic and flip-flops that drive it. In comparison, we split the circuit into sub-modules, and then encode and evaluate the power of each sub-module. This approach offers obvious advantages:

Non-overlapping: Works [6]–[8] based on logic cones unavoidably involve significant overlapping between different cones, making them actually inappropriate for power modeling. The significant logic overlap will make the summing up of the power of each cone much larger than the total design power. In contrast, our approach uses non-overlapping sub-modules defined by specific functional roles. We can accurately estimate the power of a larger component or the whole design by summing the powers of all its constituent sub-modules.

Fine-grained Analysis: By splitting based on sub-module, the ATLAS model achieves fine-grained predictions not only in terms of time but also at the component level. Since each sub-module is determined by the inherent functionality of the netlist, the power of each sub-module provides fine-grained feedback to designers.

B. Netlist Collection for Self-supervised Learning

To support the learning of the general intrinsic netlist information across stages, we will collect different netlists during preprocessing. The collected netlists will be applied in self-supervised encoder pre-training in the next step. As shown in Fig. 2, we will generate three types of netlist for model pre-training: N_g, N_p and N_g^+. The N_g is the original gate-level netlist by synthesizing RTL code. The N_g^+ and N_p are introduced below.

1) For recognition of structure and functionality: By applying logic-invariant transformations on N_g, we can derive another gate-level netlist N_g^+, which has the same functionality but a different structure. As a basic property of circuits, different gate-level netlists can implement the same functionalities. Our encoder model will learn the structural and functional similarities among sub-modules by recognizing structures with the same functionality. During encoder pre-training, as we will introduce, N_g provides original samples, while N_g^+ provides corresponding positive samples. Sub-module that differs from the target sub-module serves as a negative sample.

2) For alignment with the layout stage: We employed commercial tools to perform layout on N_g, with each layout step involving optimization. Ultimately, we obtained the post-routing design layout, and we refer to the corresponding post-layout netlist as N_p. During encoder pre-training, as we will introduce, the objective is to learn the cross-stage alignment between N_g and N_p, making the embedding of the gate-level netlist E_g to capture and reflect the information of post-layout netlist E_p.

C. Feature Collection

We split all netlists from Section III-B into sub-modules. Then we convert each sub-module to a directed graph (DG). Each node corresponds to each cell instance and directed edges are the wires connecting these nodes. By representing sub-modules as DGs, we will leverage the advantages of graph transformer models to capture the circuit structures and semantics.

After converting all sub-modules to DGs, we proceed to collect features for every graph node. To benefit the subsequent learning process, we have carefully selected four types of features: Node Type, Per-cycle Toggle, Cell Internal and Leakage Power.

1) Node Type: We categorize all cells according to their functionality into 18 types, using one-hot encoding to represent the cell type. ATLAS will learn to recognize the node type and node type effectively conveys each cell's functional characteristics.

For example, the clock tree in post-layout netlist N_p involves the interconnection of many clock-related cells, such as *Clock Buffer*, *Clock AND*, and *Clock Multiplexer*, all categorized as the `CK` type by us. By learning to recognize masked `CK`-type nodes during pre-training, the encoder is guided to capture the effect of the clock tree.

2) Per-cycle Toggle: Per-cycle toggle represents switching activity at a node's output port during a specific timestamp under real workloads. Including a per-cycle toggle aims for ATLAS to learn the signal propagation relation between nodes and to predict signal propagation. During the pre-training, the encoder will learn by predicting the masked toggle behavior.

3) Cell Internal and Leakage Power: For each cell instance, both internal and leakage power values will be parsed from the lookup tables in the `.lib` files from the technology library according to its cell type. These standard cell power values provide important cell information and are directly power-related.

IV. ATLAS Pre-training

ATLAS trains a general circuit encoder to convert each sub-module into a representation. Fig. 3 illustrates the entire self-supervised pre-training process of the encoder model based on efficient graph transformer [14]. The encoder will generate an embedding for each individual node, as well as an additional overall graph embedding for the whole input graph (i.e., corresponding sub-module).

The encoder is pre-trained on the unlabeled circuit dataset, targeting the following two general learning goals: 1) recognizing the structure, functionality, and toggle propagation of netlists; 2) learning the alignment between netlist and layout. The encoder is expected to encode the learned information into its output embedding vector. To achieve this, we carefully designed five encoder pre-training tasks: ❶ Masked toggle propagation learning; ❷ Masked node type learning; ❸ Sub-module size learning; ❹ Gate-level netlist contrastive learning; ❺ Cross-stage alignment contrastive learning.

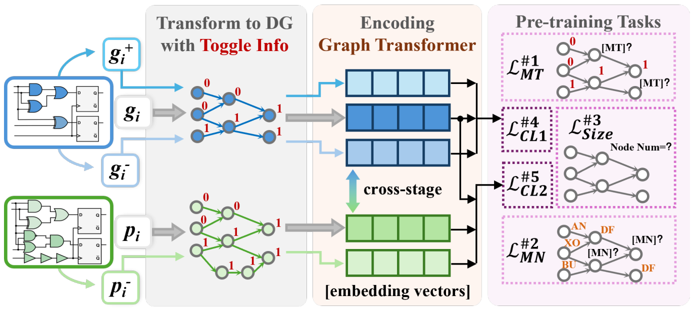


Fig. 3: ATLAS Pre-training. This framework employs a self-supervised and cross-stage flow that incorporates two types of contrastive learning. Additionally, ATLAS includes predictions for [`MASK_TOGGLE`] ([MT]) sub-module size and [`MASK_NODE_TYPE`] ([MN]).

Our proposed five self-supervised learning tasks can be roughly categorized into three types, as summarized below.

1) Masked node recovery (Tasks ❶, ❷). We will mask (i.e., hide) important node properties (e.g., type, toggle) of randomly selected nodes. A temporary MLP-based classification head will predict the masked node properties based on the encoder-generated node embedding. During pre-training, the encoder and classification head (to be discarded after pre-training) will be trained end-to-end to maximize accuracy. The encoder will thus learn to encode the structural information of connected circuit operators within the circuit graph.
2) Size recognition (Task ❸). A temporary MLP-based regression head will predict the total number of nodes in the given graph (i.e., sub-module netlist) based on encoder-generated graph embedding. During pre-training, the encoder and regression head (to be discarded after pre-training) will be trained end-to-end to maximize accuracy. The encoder will learn to encode more global graph information in the overall graph embedding.
3) Netlist contrastive learning (Tasks ❹, ❺). The encoder will be trained to minimize the embedding distance of netlists of the same sub-module with the same functionality (i.e., positive samples), while maximizing the embedding distance of different sub-modules (i.e., negative samples).

We formally introduce and formulate each learning task below.

Task #1 Masked toggle propagation learning: To train our encoder model to handle per-cycle workloads, we mask the per-cycle toggle information of randomly selected nodes. A temporary MLP classifier will predict whether the masked node is toggling based on encoder-generated node embeddings. This pre-training task trains the encoder to capture the toggle information and the propagation of toggles. Specifically, we randomly mask the toggle feature (0 or 1) of selected nodes and replace it with a special token [`MASK_TOGGLE`]. Subsequently, we predict whether these nodes will toggle based on the embedding of the masked node, which also captures neighboring nodes' information. We represent the input circuit in masked graph format as $\hat{G}$. The ground-truth toggle feature of the masked nodes is represented by $\boldsymbol{t}_{\hat{G}}^{msk}$. The predicted toggle feature for the masked nodes is represented by $\boldsymbol{p}^{msk}(\hat{G})$, which is based on node embeddings[2]. The objective is to minimize the ross-entropy (CE) loss

[2]To simplify the formulation, the encoder model does not directly appear in the loss term in Eq. (1). The prediction $\boldsymbol{p}^{msk}(\hat{G})$ is based on encoder model.

between $\boldsymbol{t}_{\hat{G}}^{msk}$ and $\boldsymbol{p}^{msk}(\hat{G})$, as expressed in the following formula:

$$\mathcal{L}_{\text{MT}}^{\#1} = \mathbb{E}_{\hat{G}\sim\mathcal{D}} CE\left(\boldsymbol{t}_{\hat{G}}^{msk},\ \boldsymbol{p}^{msk}(\hat{G})\right) \tag{1}$$

where $\mathbb{E}_{\hat{G}\sim\mathcal{D}}$ represents the expectation $\mathbb{E}$ over the circuit graph dataset $\mathcal{D}$.

Task #2 Masked node type learning: Similar to the toggle task, we randomly mask the node type information and replace it with a special token [`MASK_NODE_TYPE`]. A temporary MLP classifier will predict the one-hot encoding for the masked node types based on the embeddings of the masked node. The input circuit's masked representation is denoted as $\hat{G}$, with the ground-truth type for the masked nodes represented by $\boldsymbol{c}_{\hat{G}}^{\text{msk}}$ and the predicted type denoted by $\boldsymbol{q}^{\text{msk}}(\hat{G})$. The objective is to minimize the cross-entropy (CE) loss between the true one-hot encoding $\boldsymbol{c}_{\hat{G}}^{\text{msk}}$ and the predicted one-hot encoding $\boldsymbol{q}^{\text{msk}}(\hat{G})$, expressed as:

$$\mathcal{L}_{\text{MN}}^{\#2} = \mathbb{E}_{(E_g)\sim\mathcal{D}} CE\left(\boldsymbol{c}_{\hat{G}}^{\text{msk}},\ \boldsymbol{q}^{\text{msk}}(\hat{G}))\right) \tag{2}$$

Task #3 Sub-module size learning: The size of a circuit design is often directly correlated with its overall power consumption. We propose a pre-training task to recognize the size of each sub-module. A temporary MLP regressor will predict the number of nodes (Num_{pre}) in the whole graph (i.e., cell count in sub-module) based on the overall graph embedding. The objective is to minimize the MSE between predicted node number Num_{pre} and the actual number of nodes Num_{true}:

$$\mathcal{L}_{\text{Size}}^{\#3} = \mathbb{E}_{\hat{G}\sim\mathcal{D}}\left[(\text{Num}_{pre} - \text{Num}_{true})^2\right] \tag{3}$$

Task #4 Gate-level netlist contrastive learning: To capture the functionalities of each sub-module, we employed contrastive learning on the netlist. Specifically, for the gate-level netlist N_g with many sub-modules g_i, we generate another N_g^+ through functionally equivalent transformations. N_g^+ provides new sub-modules g_i^+ with identical functionalities but different structures. At this task, we consider g_i and g_i^+ as a pair of positive samples. All other sub-modules in the batch that exhibit different functionalities are treated as negative samples (g_i^-) of g_i. The contrastive objective pulls functionally similar sub-modules closer together in their respective embedding spaces while pushing dissimilar sub-modules further apart. We denote the embeddings of g_i, g_i^+, and g_i^- as E_g, E_g^+, and E_g^-, respectively. Specifically, we minimize the InfoNCE loss [15] used for sub-module embeddings, defined as CL, as follows:

$$\mathcal{L}_{\text{CL1}}^{\#4} = \mathbb{E}_{(E_g)\sim\mathcal{D}} CL\left(E_g, E_g^+, E_g^-\right) \tag{4}$$

Task #5 Cross-stage alignment contrastive learning: For cross-stage contrastive learning, similar to Task #4, we aim for gate-level netlist N_g to be closer in the embedding space to the corresponding post-layout netlist N_p. This alignment task targets capturing layout information from N_p in netlist embedding. For such contrastive learning, the original sample is still g_i, while the positive sample is p_i, and the negative samples p_i^- comprise all other functionally distinct sub-modules in the batch. We denote the embeddings of p_i and p_i^- as E_p and E_p^-, respectively. We also adopt the InfoNCE loss function for this purpose:

$$\mathcal{L}_{\text{CL2}}^{\#5} = \mathbb{E}_{(E_g,E_p)\sim\mathcal{D}} CL\left(E_g, E_p, E_p^-\right) \tag{5}$$

Finally, we formulate the complete self-supervised pre-training objective of our model by jointly employing the five tasks:

$$\mathcal{L} = \mathcal{L}_{\text{MT}}^{\#1} + \mathcal{L}_{\text{MN}}^{\#2} + \mathcal{L}_{\text{Size}}^{\#3} + \mathcal{L}_{\text{CL1}}^{\#4} + \mathcal{L}_{\text{CL2}}^{\#5} \tag{6}$$

Regarding the encoder of ATLAS, traditional Graph Neural Networks (GNNs) [16] and Graph Transformers [17] [18] can be resource-intensive when applied to large-scale graphs. Considering that some of our DGs contain tens of thousands of nodes, we employ an efficient Graph Transformer, SGFormer [14], as the encoder for ATLAS. In our DGs, SGFormer achieves propagation capabilities across any node using a single global attention network of linear complexity, while maintaining linear complexity with respect to the number of nodes. Furthermore, it does not require positional encodings, feature or graph preprocessing, or additional loss functions.

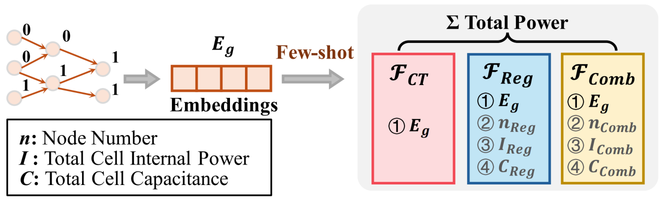


Fig. 4: ATLAS Fine-Tuning. The total post-layout power is divided into different groups as three separate fine-tuning tasks, which are predicted by three fine-tuning models. Other than embedding vectors E_g, ATLAS uses cell internal power, cell capacitance, and node number as features, avoiding the need for cumbersome feature engineering and complex machine learning models.

V. ATLAS Fine-tuning

To apply pre-trained ATLAS on downstream power modeling tasks, as shown in Fig. 4, we divide the total power into three power groups: the clock tree power, the combinational logic power, and the register power[3]. Each power group will be modeled by its respective fine-tuning model, denoted as $\mathcal{F}_{CT}$, $\mathcal{F}_{Comb}$, and $\mathcal{F}_{Reg}$. These fine-tuned power models differ from purely supervised methods by utilizing the embedding E_g from the encoder as features, rather than relying on cumbersome feature engineering and complex ML models to process raw circuits.

Regarding the clock tree group, it is completely absent in the post-synthesis netlist N_g and appears only in post-layout netlist N_p, which is unknown during inference. Therefore, we do not incorporate any additional features in addition to the embedding. We rely solely on the embedding E_g to predict the power of the clock tree group. This approach will demonstrate that our encoder model has effectively learned the alignment between N_g and N_p.

For both combinational and register groups, we choose to combine the embeddings E_g with additional relevant features available from netlist N_g. For the combinational group, we utilize the number of combinational nodes n_{Comb} of the post-synthesis netlist, along with the total cell internal power I_{Comb} and total cell capacitance C_{Comb} of the combinational group as features. The cell internal power and cell capacitance of each node are collected from the lookup tables in the `.lib` file of the technology library. To calculate the total cell internal power I_{Comb} and total cell capacitance C_{Comb}, we multiply each cell internal power and capacitance in the combinational section by the per-cycle toggle of their output pins and sum up the results.

As for the register group, similar to the combinational group, we adopt the number of register nodes n_{Reg}, the total cell internal power I_{Reg} and total cell capacitance C_{Reg} of the register group from N_g as additional features for fine-tuning.

As a result, our fine-tuning models are lightweight and easily integrable, including tree-based models, such as XGBoost [19]. So

[3]Our register power group includes (and is dominated by) the power of the clock pin in each register. Accordingly, our clock tree power group does not include such register clock pin power.

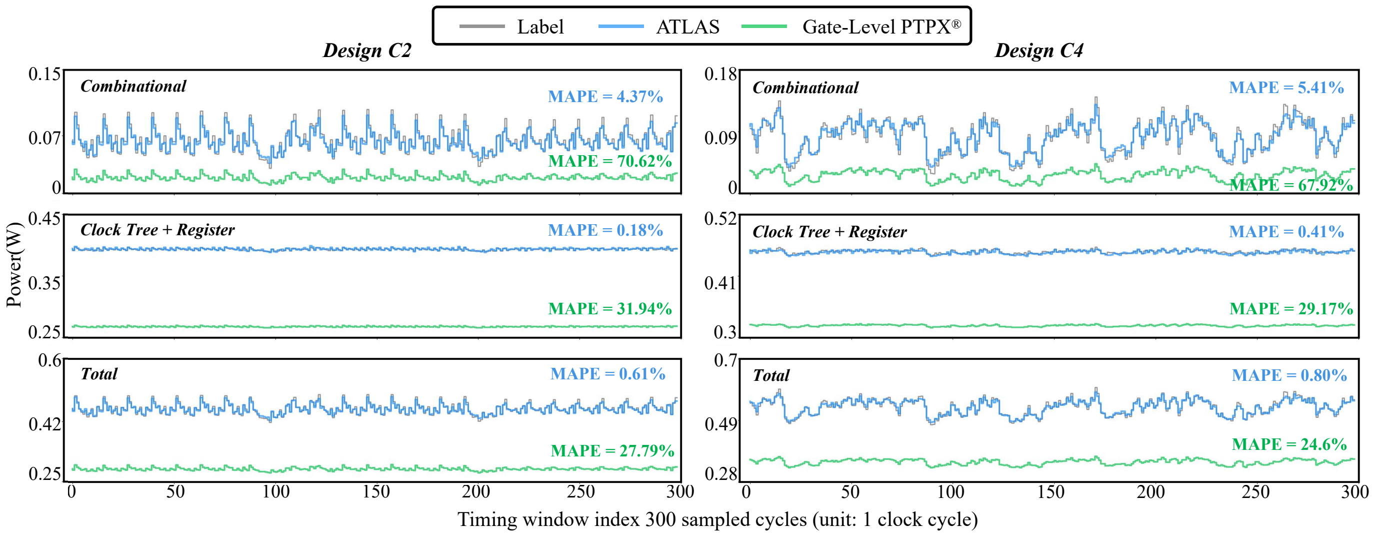


Fig. 5: Power prediction results from ATLAS across 300 cycles for *C2* and *C4* under W_1. The labels were obtained by simulating the netlist and SPEF files using PTPX® after detailed routing with Innovus®. The Gate-Level PTPX® values, which are obtained from the gate-level netlist, illustrate the significant differences in netlist power between gate level and layout.

the total predicted power is:

$$P_{total} = \mathcal{F}_{CT}(E_g) + \mathcal{F}_{Comb}(E_g, n_{Comb}, I_{Comb}, C_{Comb}) + \mathcal{F}_{Reg}(E_g, n_{Reg}, I_{Reg}, C_{Reg}) \tag{7}$$

VI. Experimental Results

A. ATLAS Implementation and Experiment Data Generation

We implement our ATLAS using PyTorch and PyTorch Geometric (PyG), a library built on top of PyTorch that facilitates writing and training GNNs. Our pre-training is conducted on a Linux machine equipped with an NVIDIA A6000 GPU, while the fine-tuning tasks are performed on an Intel Xeon processor with 128GB of memory.

RTL simulation on workloads is based on Synopsys VCS®[20] (VCS). The logic synthesis is executed at a clock frequency of 1GHz using Synopsys Design Compiler®[21] (DC). In our experiments, we utilized the TSMC 40nm standard cell library [22], along with the corresponding Memory Compiler. We utilized the Innovus® to perform mixed-size placement, clock tree synthesis, and routing, with each step including timing optimization, ultimately resulting in the chip design layout. After detailed routing with Innovus®, we dump out the post-layout ultimate netlist and corresponding RC values in Standard Parasitic Extraction Format (SPEF) files. We conduct accurate layout-stage power simulation based on the post-layout netlist, SPEF file, and different workloads (denoted as W_1, W_2). The per-cycle ground-truth layout power simulation is performed based on Synopsys PrimeTime PX® [1] (PTPX®).

In this work, we generated a dataset with six different realistic designs (named C1 to C6 from smallest to largest) that support workload simulation. Table II presents the cell count statistics for six different designs at both the gate-level and post-layout stages. The cell counts range from approximately 300,000 to 600,000, indicating their difference. For the same design, the increase in cell count from the gate-level stage to the post-layout stage reflects the impact of timing optimization and clock tree synthesis during the layout process.

For ATLAS pre-training stage, our dataset comprises aligned 3,253 DG pairs of N_g, N_g^+, N_g^- and N_p for each cycle. We employed four designs (*C1*, *C3*, *C5* and *C6*) for training and two designs (*C2* and *C4*) for testing, ensuring absolute no overlap between the training and testing designs. Ultimately, the number of DGs in the training and testing datasets is 92,500 and 37,640, respectively. During the encoder pre-training process, the five self-supervised tasks are trained simultaneously for 60 epochs, requiring approximately 21 hours. Despite such training runtime, please notice that the encoder only needs to be pre-trained once and then applied to different designs. We employ ReLU as the activation function and set the batch size to 16, with a learning rate of 1e-4 using the Adam optimizer. In the fine-tuning phase, we use XGBoost with 500 estimators and a depth of 5, taking only several seconds for training.

	C1	*C2*	*C3*	*C4*	*C5*	*C6*
Gate-level	289384	322664	389120	399486	465129	597877
Post-layout	301650	340923	412186	422391	494614	638666

TABLE II: The statistics of the gate counts for the six designs at the gate-level and post-layout stages.

B. Power Modeling Experiment Setup

Commericial Tool as Baseline. Our experiment will evaluate the modeling accuracy and efficiency of per-cycle post-layout power based on post-synthesis netlists (without any layout information). Moreover, the tested design is strictly not *seen* by the power model during training. As summarized in the introduction, prior works either require a design-specific model for time-based power modeling [3], [4] or can only evaluate the average power [5]–[9]. As a result, no prior ML-based power models apply to this challenging cross-design per-cycle power modeling task. Moreover, it is very challenging to naturally adapt prior works to our brand-new tasks not supported originally. Therefore, we choose to compare ATLAS with the standard commercial tool at the post-synthesis netlist stage.

Exclusion of Memory Group from Results. In our analysis of power consumption, memory (i.e., SRAM) accounts for almost half of the total power in the entire design. But for this memory power group, we can predict its power with very high accuracy, without too much effort at the netlist stage. Since the SRAM macro is unchanged during layout, we developed a basic ML model based on the toggle activities of memory ports from workload, and energy values from lookup tables in SRAM `.lib` files. Our model achieves an error of only 0.5% without relying on any power simulators. Given such high accuracy in memory power prediction, incorporating the memory power model into ATLAS would lead to a lower error, but primarily dominated by the memory group. To demonstrate the benefits of our ATLAS effectively, we choose to exclude this easier memory group in results. Instead, we focus on predicting the more challenging other powers of the combinational, clock tree, and register groups.

Design & Workload		Error Percentage of ATLAS					Error Percentage of Gate-Level PTPX®				
		Combinational	Clock Tree	Register	Clock Tree + Register	Total	Combinational	Clock Tree	Register	Clock Tree + Register	Total
C2	W_1	4.37%	0.17%	0.27%	0.18%	0.61%	70.62%	100%	2.36%	31.94%	27.79%
	W_2	5.35%	0.15%	0.32%	0.22%	0.57%	71.01%	100%	2.22%	31.93%	27.81%
C4	W_1	5.41%	1.07%	0.54%	0.41%	0.80%	67.92%	100%	2.34%	29.17%	24.60%
	W_2	5.34%	0.93%	0.68%	0.67%	1.13%	69.36%	100%	2.44%	29.23%	25.08%
Average		5.12%	0.58%	0.45%	0.37%	0.78%	69.73%	100%	2.34%	30.57%	26.32%

TABLE III: MAPE (%) of design *C2* and *C4* under two workloads W_1 and W_2.

As introduced Section V, we implemented three corresponding fine-tuned models: $\mathcal{F}_{CT}$, $\mathcal{F}_{Comb}$, and $\mathcal{F}_{Reg}$.

We evaluate the accuracy of per-cycle power modeling with Mean Absolute Percentage Error (MAPE) between label Y_i and prediction $\hat{Y}_i$, assuming altogether m cycles in the tested workload.

$$\text{MAPE} = \frac{1}{m}\sum_{i=1}^{m}\left|\frac{Y_i - \hat{Y}_i}{Y_i}\right| \times 100\% \tag{8}$$

C. Power Prediction Results

Fig. 5 visualizes power prediction results (ATLAS) and the actual power values (Labels) over 300 cycles for testing designs *C2* and *C4*, under workload W_1. For comparison, we present the commercial tool's simulation result on exactly the same gate-level netlist and workload (Gate-Level PTPX®). For both *C2* and *C4*, the overall power trace shapes of ATLAS across the three power groups closely resemble the labels, with the total MAPE being only 0.61% and 0.80%, respectively. In comparison, the gate-level PTPX® error is higher than 25% for both designs. Its power trace also differs significantly from the label. This error shows the gap between the power of the gate-level netlist and the power of the ultimate layout.

In Table III, we provide a more comprehensive comparison of results for ATLAS and Gate-Level PTPX®, which displays the MAPE for different power groups under both workloads W_1 and W_2. Gate-Level PTPX® still exhibits the $> 25\%$ prediction accuracy in total power in all cases, while ATLAS shows $< 1\%$ error.

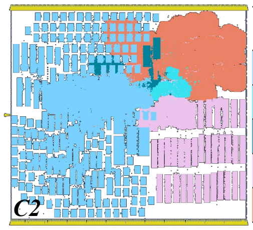
In terms of the most challenging combinational group prediction, ATLAS shows an average MAPE of 5%, while Gate-Level PTPX® has a considerably higher MAPE of 69%. Since there is no clock tree in the gate-level netlist, Gate-Level PTPX® shows a MAPE of 100%. ATLAS demonstrates a remarkably low MAPE of only 0.58%.

In summary, standard commercial tools at the gate-level stage show $> 25\%$ error on time-based layout power due to the lack of layout information. ATLAS, by pre-training and fine-tuning, well captures the potential impact of layout and demonstrates a remarkable $< 1\%$ error for the per-cycle power of new circuit design.

Fig. 6 further illustrates the ATLAS-predicted power of five major components in the design $C2$. The prediction of each component's power is the summation of the predicted power of all its sub-modules. Combined with the layout, the component power predictions further provide power distributions on the layout. In the table on the right of Fig. 6, we present each component's labels, ATLAS power predictions, and the associated MAPE values. It is evident that the power predictions vary across different components, with some over-estimation and others under-estimation. The component error is slightly higher than total power but mostly maintained $< 5\%$.

D. Runtime Comparison

Table IV presents the time taken for back-end processing using Innovus, the time for power simulations using PTPX®, and the time taken by ATLAS to directly predict power. Here we included the runtime used for the entire preprocessing step in ATLAS. We calculated the average time (in seconds) for six designs. Compared with the complete traditional flow that performs both complete layout and time-based power simulations, ATLAS finished estimating the 300-cycle workload with $> 1000\times$ less runtime.



C2 **Component**	**Label (*W*)**	**ATLAS (*W*)**	**MAPE (%)**
frontend	0.1894	0.1898	1.01
lsu	0.0217	0.0229	5.59
ptw	0.0094	0.0092	2.07
dcache	0.0551	0.0551	1.44
core	0.1821	0.1791	1.68

Fig. 6: Component-level power analysis of *C2* under W_1, illustrating the division into five major components, each comprised of their respective sub-modules, highlighting ATLAS's accuracy in the component level. C2 is an out-of-order CPU design, its main components include the CPU frontend, load-store unit (LSU), data cache (dcache), etc.

Design	**ATLAS**			**Traditional Flow**		
	Pre.	**Infer**	**Total**	**P&R**	**Simulation**	**Total**
C1	62	3.2	65.2	46392	96	46488
C2	66	4.8	70.8	55327	98	55425
C3	67	4.1	71.1	69624	105	69709
C4	72	4.6	76.6	79836	112	79948
C5	75	5.1	80.1	105742	131	105873
C6	86	4.9	90.9	124860	156	125016
Average	72	4	**76**	80297	116	**80413**

TABLE IV: Runtime (in seconds) comparison for 300 cycles extracted from W_1. Column names represent: Pre. (Data Preprocessing), Infer (Inference), Simulation (Time-based Power Simulation).

In summary, compared to standard design flow based on commercial tools, ATLAS offers three significant advantages: ❶ it eliminates the need for the time-consuming layout process; ❷ it reduces the time required for per-cycle power simulation; ❸ the accuracy of power predictions for post-layout netlists by ATLAS far exceeds that of Gate-Level PTPX® at the early-stage of gate-level netlists.

VII. Conclusion

In this paper, we propose ATLAS, a pioneering framework for fine-grained per-cycle power prediction. ATLAS is the first work that supports both time-based power simulation and general cross-design power modeling. It achieves unprecedented general time-based power modeling based on a customized pre-training and fine-tuning paradigm. When evaluated on the prediction of per-cycle post-layout power based on gate-level netlist, ATLAS achieves a remarkable MAPE of $< 1\%$ for total power estimation with inference speeds that are $> 1000\times$ faster than traditional flow.

VIII. Acknowledgments

This work is supported by National Natural Science Foundation of China (NSFC) 62304192, Hong Kong Research Grants Council (RGC) ECS Grant 26208723, and ACCESS–AI Chip Center for Emerging Smart Systems, sponsored by InnoHK, Hong Kong SAR. We thank HKUST Fok Ying Tung Research Institute and National Supercomputing Center in Guangzhou Nansha Sub-center for computational resources.

References

[1] Synopsys, "PrimePower: RTL to signoff power analysis," 2023. [Online]. Available: https://www.synopsys.com/implementation-and-signoff/signoff/primepower.html

[2] Siemens, "PowerPro® RTL Low-Power," https://eda.sw.siemens.com/en-US/ic/powerpro/, 2021.

[3] Y. Zhou, H. Ren, Y. Zhang, B. Keller, B. Khailany, and Z. Zhang, "Primal: Power inference using machine learning," in *DAC*, 2019.

[4] Z. Xie *et al.*, "Apollo: An automated power modeling framework for runtime power introspection in high-volume commercial microprocessors," in *MICRO*, 2021.

[5] P. Sengupta, A. Tyagi, Y. Chen, and J. Hu, "How good is your Verilog RTL code? A quick answer from machine learning," in *International Conference on Computer-Aided Design (ICCAD)*, 2022.

[6] C. Xu, C. Kjellqvist, and L. W. Wills, "Sns's not a synthesizer: a deep-learning-based synthesis predictor," in *ISCA*, 2022.

[7] C. Xu, P. Sharma, T. Wang, and L. W. Wills, "Fast, robust and transferable prediction for hardware logic synthesis," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 167–179.

[8] 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 *ICCAD*, 2023.

[9] Y. Du, Z. Guo, X. Jiang, Z. Chai, Y. Zhao, Y. Lin, R. Wang, and R. Huang, "Powpredict: Cross-stage power prediction with circuit-transformation-aware learning," in *DAC*, 2024.

[10] Y. Zhang, H. Ren, and B. Khailany, "Grannite: Graph neural network inference for transferable power estimation," in *DAC*, 2020.

[11] M. Li, S. Khan, Z. Shi, N. Wang, H. Yu, and Q. Xu, "Deepgate: learning neural representations of logic gates," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022.

[12] Z. Shi, H. Pan, S. Khan, M. Li, Y. Liu, J. Huang, H.-L. Zhen, M. Yuan, Z. Chu, and Q. Xu, "Deepgate2: Functionality-aware circuit representation learning," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023, pp. 1–9.

[13] 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 *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022.

[14] Q. Wu, W. Zhao, C. Yang, H. Zhang, F. Nie, H. Jiang, Y. Bian, and J. Yan, "Sgformer: Simplifying and empowering transformers for large-graph representations," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[15] A. Oord, Y. Li, and O. Vinyals, "Representation learning with contrastive predictive coding," 07 2018.

[16] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *International Conference on Learning Representations*, 2019.

[17] Q. Wu, C. Yang, W. Zhao, Y. He, D. Wipf, and J. Yan, "DIFFormer: Scalable (graph) transformers induced by energy constrained diffusion," in *The Eleventh International Conference on Learning Representations*, 2023.

[18] Q. Wu, W. Zhao, Z. Li, D. Wipf, and J. Yan, "Nodeformer: a scalable graph structure learning transformer for node classification," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, 2024.

[19] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2016.

[20] "VCS® functional verification solution," https://www.synopsys.com/verification/simulation/vcs.html, 2021.

[21] "Design Compiler® RTL Synthesis," https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-nxt.html, 2021.

[22] *TSMC 40nm LP process technology*, https://www.tsmc.com/ english/dedicatedFoundry/technology/logic/l_40nm, 2008.