Timing Estimation and Optimization for Physical Design Using Machine Learning Approaches

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Wenjing Jiang

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Sachin S. Sapatnekar

January, 2025

© Wenjing Jiang 2025
ALL RIGHTS RESERVED

Acknowledgements

I would like to express my deepest gratitude to my advisor, Professor Sachin S. Sapatnekar, for his guidance, patience, and motivation throughout my graduate studies. Over the past six years, it has been both an honor and a privilege to work under his mentorship. The Ph.D. journey has left me with invaluable academic experience. His steadfast dedication to science, profound insights, and perspective on life have profoundly influenced not only my academic growth but also my personal outlook and career trajectory.

I am also very grateful to my committee members Prof. Kia Bazargan, Prof. Changhyun Choi, and Prof. Antonia Zhai for their valuable and constructive feedback. I also extend my heartfelt thanks to all my collaborators. I am very grateful to Prof. Andrew B. Kahng from UC San Diego for his guidance and the opportunity to work with him on the OpenROAD project. The experience was helpful in shaping my research and professional development. I would also like to thank Dr. Jin Yan and my colleagues at Google for their guidance and support during my internship, as well as Dr. Haiguang Liao and Dr. Prabal Basu at Cadence for sharing their expertise in the EDA field and providing invaluable technical insights during my internship at Cadence, which significantly enriched my research. Each of these experiences has contributed significantly to my academic and professional journey.

Beyond academic pursuits, a Ph.D. is a profound period of self-discovery and growth. I cherished the natural beauty and outdoor adventures in Minnesota. My labmates have been an essential part of this journey, creating a vibrant and supportive environment that I will always treasure. I would like to thank Dr. Vidya A. Chhabria, Dr. Meghna Mankalale, Dr. Zhaoxin Liang, Dr. Qianqian Fan, Dr. Tengtao Li, Dr. Masoud Zabihi, Dr. Tonmoy Dhar, Dr. Susmita Dey Manasi, Dr. Kishor Kunal, Dr. Sudipta Mondal, Dr. Nibedita Karmokar, Ziqing Zeng, Mohammad Shohel, Abhimanyu Kumar, Hangyu Zhang, Subhadip Ghosh, Divya Yogi, Endalkachew Gebru, Ziyang Jiang, and Sosina

Berhan. The time spent with this group of people has made my Ph.D. journey more enriching and unforgettable. To my partner, Dr. Sihan Xie, I owe my deepest thanks for his unwavering support and encouragement at every step of this journey. His presence has brought color, joy, and strength to my life, making every challenge more manageable and every success more meaningful.

Lastly, I would like to express my love and gratitude to my parents, Fang Deng and Chuanbin Jiang, and my grandmother, Yin Fang. Their love, unwavering support, and faith in my decisions have been my greatest sources of strength. They have stood by me through every difficulty and celebrated every milestone, and for this, I am forever grateful.

Dedication

To my family, whose support has made this journey possible.

Abstract

Rapid progress in semiconductor technology has driven integrated circuit (IC) designs to become increasingly complex, resulting in significant challenges in achieving optimality in design. The growing intricacy and the high design costs of modern ICs demand accurate prediction of the quality of results (QoR) to guide early design decisions. Inaccurate predictions can lead to inefficient design iterations, degraded QoR, and even design failures. Therefore, improving QoR prediction accuracy while maintaining computational efficiency has become a critical objective in electronic design automation (EDA). The recent advancements in machine learning (ML) have provided promising solutions for addressing these challenges. ML-based predictive models and optimization methodologies have been developed to improve both quality and efficiency across the design flow.

The first part of the thesis focuses on timing prediction after placement and clock tree synthesis. Due to the unavailability of detailed routing information in design stages prior to detailed routing (DR), the tasks of timing prediction and optimization pose major challenges. This part first documents that having "oracle knowledge" of the final post-DR parasitics enables post-gloabal routing(GR) optimization to produce improved final timing outcomes. To bridge the gap between post-GR timing estimation and final timing results during post-GR optimization, ML-based parasitic and interconnect delay models are proposed for accurate path delay estimation. These models, trained on diverse datasets, demonstrate higher prediction accuracy compared to traditional methods based on the route guide generated in GR stage. Applied during post-GR optimization, the design shows better timing slack in post-DR without exacerbating routing congestion. The methodology is applied to both open-sourced tool flows and a commercial tool flow. The results on an open-source 45nm bulk and a commercial 12nm FinFET enablement show the robustness and good generalization of the proposed models under varying clock constraints and noisy training data.

The second part of the thesis focuses on engineering change orders (ECOs) in late design stages, where minimal design fixes are required to address the timing shifts caused by excessive IR drops. We integrate IR-drop-aware timing analysis and reinforcement learning (RL) to develop an efficient ECO timing optimization. The method operates after physical design and power grid synthesis, and rectifies IR-drop-induced timing

degradation through gate sizing. It incorporates the conventional gate sizing technique, Lagrangian relaxation (LR), into a novel RL framework, which trains a relational graph convolutional network (R-GCN) agent to sequentially size gates to fix timing violations. The R-GCN agent outperforms a classical LR-only algorithm in an open 45nm technology. It moves the Pareto front of the delay-power tradeoff curve to the left, saves runtime over the prior approaches by running fast inference using trained models, and reduces the perturbation to placement by sizing fewer cells. It is also shown to be transferable across timing constraints and adaptable to unseen designs with fine-tuning, further highlighting its versatility and efficiency.

The last part of the thesis studies the correlation between proxy metrics used in traditional logic optimization and actual post-synthesis delay, and the importance of accurate timing estimation on the effectiveness of logic optimization. As circuit designs become more intricate, obtaining accurate performance estimation in early stages, for effective design space exploration, becomes more time-consuming. Traditional logic optimization approaches often rely on proxy metrics to approximate post-synthesis performance and area. However, these proxies do not always correlate well with actual post-mapping delay and area, resulting in suboptimal designs. To address this issue, a ground-truth-based optimization flow is explored to directly incorporate the exact post-synthesis delay and area during optimization. While this approach improves design quality, it also significantly increases computational costs due to finishing technology mapping for every logic optimization iteration, particularly for large-scale designs. To overcome the runtime challenge, we apply ML models to predict post-mapping delay and area using the features extracted from logic represenation graph. Our experimental results show that the model has high prediction accuracy with good generalization to unseen designs. Furthermore, the ML-enhanced logic optimization flow significantly reduces runtime while maintaining comparable performance and area outcomes.

Contents

Acknowledgements **i**

Dedication **iii**

Abstract **iv**

Contents **vi**

List of Tables **ix**

List of Figures **xi**

1 Introduction **1**

1.1 Challenges and opportunities in modern design 1

1.2 Thesis contributions and organization 3

2 Preliminaries **6**

2.1 Physical design and static timing analysis 6

2.1.1 Parasitic estimation for timing optimization 7

2.1.2 Static timing analysis (STA) and delay models 9

2.2 Machine learning approaches 11

2.2.1 Ensemble decision tree models 11

2.2.2 Graph neural networks 11

2.2.3 Reinforcement learning methods 13

3 Bridging the timing Gap between Global Route and Detailed Route Using ML **17**

3.1 Introduction 17

3.2 Preliminaries 24
3.2.1 Post-GR and post-DR routing estimates 24
3.3 DR timing prediction framework 25
3.3.1 Timing prediction in routing flow 25
3.3.2 ML engine 27
3.3.3 Feature engineering 27
3.4 Model training and inference in the physical design flow 30
3.4.1 Ground-truth data generation and model training 30
3.4.2 ML inference in physical design flow 31
3.4.3 Feature sensitivity 32
3.5 Experimental setup and evaluation 34
3.5.1 Model accuracy evaluation 36
3.5.2 Impact on post-DR outcomes 41
3.5.3 Generalization to different clock periods 45
3.5.4 Noise impact on model performance 48
3.6 Conclusion 49

4 IR-Aware ECO Timing Optimization Using Reinforcement Learning 50
4.1 Introduction 50
4.2 Problem formulation 52
4.3 RL-LR ECO timing framework 53
4.3.1 The circuit netlist as an annotated graph 53
4.3.2 Mapping IR-aware gate sizing to RL 54
4.3.3 Application of the R-GCN agent 57
4.4 RL model training 57
4.4.1 Core training strategy 57
4.4.2 Problem-specific training enhancement 59
4.5 Evaluation of RL-LR ECO 61
4.5.1 Experimental setup 61
4.5.2 Optimization and delay-power tradeoffs 62
4.5.3 Importance of LR-based weight adaptation 65
4.5.4 Transferability across designs 65
4.6 Conclusion 66

5 ML-based AIG Timing Prediction to Enhance Logic Optimization **67**
5.1 Introduction 67
5.2 Background 69
5.2.1 Limitations of optimizing proxy metrics 70
5.2.2 Performance-driven logic optimization 70
5.3 Methodology 72
5.3.1 ML-enhanced logic optimization 72
5.3.2 Feature engineering 73
5.3.3 Data generation and model training 77
5.4 Experimental setup and results 77
5.4.1 Evaluation of model accuracy 78
5.4.2 Evaluation of ML-enhanced logic optimization 78
5.5 Conclusion 80

6 Conclusion **81**

References **83**

List of Tables

3.1 Comparison of post-DR WS when using GR-based vs. DR-based parasitics for post-GR timing optimizations. 20
3.2 Summary of designs implemented in 45nm, 12nm, and 130nm technology nodes. 31
3.3 Prediction error of the shared ML model, and the separate ML models for designs with macros and designs without macros in 45nm. 36
3.4 Evaluation of the ML models using mean, maximum, and standard deviation of the absolute %error as metrics. 37
3.5 Impact of ML-based prediction for designs with macros on post-DR metrics for traditional, ground-truth and ML-based flows in OpenROAD. . 41
3.6 Impact of ML-based prediction on post-DR metrics for traditional, ground-truth and ML-based flows in a commercial tool. 42
3.7 Mean %error of prediction for designs generated using different clock periods. 46
3.8 Impact of training-stage noise, for various values of the noise standard deviation, on prediction accuracy. 47
3.9 Impact of noise with various standard deviation on post-DR results. . . 47
4.1 List of the annotated features on instance i in G. 54
4.2 IR-induced Timing Failures 62
4.3 ECO results for three flows: LR baseline, inference across timing constraints (Inf-TC), and RL-LR ECO training 63
4.4 Results on multiple clock constraints for wb_conmax 64
4.5 Transferability across designs 66
5.1 Post-mapping performance for two AIGs with the same number of levels and nodes 70
5.2 Features extracted from the AIG 76

5.3 Accuracy of XGBoost model for timing prediction 78
5.4 Runtime for the three flows 79

List of Figures

1.1 The rising costs of design with each new technology node. [1] 2

2.1 Timing optimizations in the physical design flow with different parasitic estimates. In modern production flows, the output of vendor A's synthesis tool is "de-buffered" when passed to vendor B's place-and-route (P&R) tool. Then, ERC compliance is enforced during global placement with buffering and resizing. The de-buffering and ERC-fixing steps are respectively marked with (*) and (**) in the figure. The wireload model is used to estimate parasitics for gate-level netlist. In placement and CTS stage, FLUTE Steiner tree is used. In global routing (GR) stage, GR Steiner tree that follows the route guide is used. After detailed routing, design rule checks (DRCs) are performed to ensures the layout adheres to the manufacturing process rules, and parasitic information are extracted into SPEF file. The worst negative slack (WNS) and total negative slack (TNS) are two metrics for timing analysis. 8

2.2 Timing paths in a design example and the RC tree model of a net in a logic stage, and its reduction to an equivalent π-model. 9

2.3 Multi-layer GCN with C input channels and F feature maps in the output layer. X_is is the node features vectors of node i at the input layer, and Z_i is the node feature vector of node i at the output layer. The graph structure is shared over layers, where edges are shown as black lines. The labels of nodes are denoted by Y_i [2]. 12

2.4 Schematic of the RL paradigm, showing interactions between an RL agent and a system. 13

3.1 Discrepancy between post-GR and post-DR wire delays of bp_be 45nm and swerv_wrapper 45nm in OpenROAD (left figures) and bp_be 45nm and swerv_wrapper 45nm in a commercial tool flow (right figures). The slope k and the intercept b in the best-fit $y = kx + b$ (red traces) differ across tool flows and across designs. 18
3.2 Net detours due to macro blockage in swerv_wrapper, 45nm. Total capacitance, total resistance and wire delay of source-sink1 in the sample net. 21
3.3 The effect of different buffer locations on the routing results: (a) The inserted buffer is placed in a different row of the NAND gate and OR gate. (b) The inserted buffer is placed in the same row as the NAND gate and OR gate. 22
3.4 Differences between the GR guides and post-DR routes of a 5-pin net from swerv_wrapper 45nm. (a) Route guides in post-GR. (b) Post-DR routes. (c) Net highlighted in congestion map. 24
3.5 Flows that use different parasitic estimates for post-GR timing optimizations: (a) traditional flow (Steiner tree-based RC estimates), (b) ground-truth (DR followed by parasitic extraction to determine post-DR parasitics), and (c) our flow (fast ML engine for post-DR parasitics and timing prediction). 25
3.6 A route guide for a three-pin net with a source and two sinks. (a) The net is routed using four wire segments with lengths l_1, on M1; l_2, on M2; l_3, on M1; and l_4, on M2. (b) The net traverses two GCells that are 80% blocked by a macro and two GCells that are 40% blocked and 20% blocked. 27
3.7 Feature sensitivity analysis showing the average mean absolute percentage error (MAPE) (y-axis), for each removed feature (x-axis), for all 45nm designs and all 12nm designs. 33
3.8 Macro feature sensitivity for wires that have area in a bounding box that with >80% blockage. 34
3.9 Wire delay and slew comparisons for swerv_wrapper 45nm. 38
3.10 Wirelength distribution of swerv_wrapper 45nm: (a) for a design with size of $1.20 \times 1.09mm^2$ generated in OpenROAD, and (b) for a design with size of $1.18 \times 1.18mm^2$ generated in a commercial tool flow. 39
3.11 Path slack comparison for (a) swerv_wrapper 45nm and (b) coyote 12nm. 40

3.12 A critical path from bp_be 45nm after DR, from the traditional flow (left) and from the ML-based flow (right). 43
3.13 Number of post-DR congested regions in the traditional and ML-based flows in (a) 45nm designs and (b) 12nm designs. 44
3.14 Sensitivity of wire delay models to different clock periods for (from left to right) swerv_wrapper, bp_fe, bp_be in 45nm, and swerv_wrapper, coyote in 12nm. 45
4.1 State and action spaces from the circuit netlist. 54
4.2 R-GCN aggregation from a 3-hop neighborhood. 56
4.3 Structure of the three-layered R-GCN agent. 57
4.4 Overview of the RL-based ECO gate sizing flow. 58
4.5 Results of training, showing the convergence of the WNS and TNS. . . . 60
4.6 Power-delay tradeoff curves for wb_conmax in a 45nm technology, showing a shift to the right due to IR drop, and the results of our RL-LR ECO approach. 61
4.7 Power overhead savings for wb_conmax. 64
5.1 Scatter plot: post-mapping circuit delay vs. the number of AIG levels. . 69
5.2 Runtime comparison for one iteration in the original logic optimization flow and the ground-truth-based logic optimization flow. The x-axis shows the name of each designs, with the number of its AIG nodes in parentheses. 71
5.3 Three flows for AIG optimization. 72
5.4 Feature extraction in an example AIG. (a) The maximum depth is annotated for each PO. (b) The fanout of each node is annotated as the weight and the updated maximum depth is annotated for each PO. (c) The binary-encoded fanout of each node is annotated as the weight and the updated maximum depth is annotated for each PO. (d) The subgraph of PO1 is highlighed in blue, and the number of paths is annotated for each PO. 74
5.5 A comparison of the Pareto-optimal fronts for the delay and area of a test design from the baseline, ground-truth-based, and ML-based flows. 79

Chapter 1

Introduction

1.1 Challenges and opportunities in modern design

Modern IC design is marked by increased functionality, higher performance requirements, and reduced power budgets, all within the constraints of shrinking technology node and escalating design complexity. As semiconductor technology approaches physical limits, the cost of design at advanced nodes has risen dramatically.

Challenges in scaling and design cost

The goal of the semiconductor industry is to be able to continue to scale the technology to improve overall performance at reduced power and cost. Nowadays, most high-performance chips are being designed in the most leading-edge process nodes. However, the continuation of Moore's law through lateral scaling has become difficult, particularly beyond the 5nm and 3nm technology nodes. The semiconductor industry faces challenges in technology scaling including power scaling, parasitics scaling, interconnect scalability, wafer processing cost. To address these issues, a range of innovations is being explored to address these challenges such as new device architecture for low-power design and 3D integration/stacking for more dense high-performance design with shorter interconnect, which are expected to bring performance, power, and area (PPA) gain for future designs [3]. Despite these advancements, the industry faces a significant challenge as the cost of designing new products at advanced technology nodes has skyrocketed. Fig. 1.1 shows the rising costs of SoC designs in new technology nodes. The design costs of a complex 5nm SoC are over 80% higher than the design cost of a 7nm SoC. This underscores the critical need to achieve the design specification within the constraints of existing design resources while designing in advanced technology nodes. Therefore,

improving design efficiency has gained much attention. In the long design process, the estimation on the quality of the result (QoR) in each design stage plays an important role in accelerating the design process and improving the quality of the final design result.

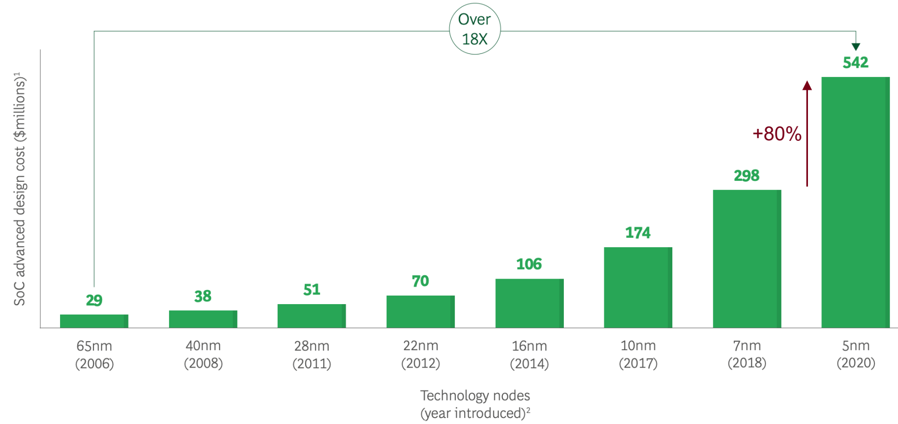


Figure 1.1: The rising costs of design with each new technology node. [1]

Challenges in predicting the quality of results of design, and the role of ML Achieving design convergence in modern ICs is full of challenges, many of which stem from the intricate interplay of design parameters. For instance, between the placement and timing closure, achieving timing closure requires accurate path delays, which depend on placement and routing. However, early placement decisions significantly affect routing congestion and path delays. Timing estimates are used to drive placement, but inaccuracies in these estimates can lead to suboptimal placements, requiring numerous iterative steps before they can be resolved. Another example of the cyclical dependence between routing and signal integrity also shows this kind of challenge. Signal integrity issues such as crosstalk and noise depend on routed nets, but routing decisions depend on signal integrity requirements: the crosstalk and noise information become apparent only after nets are routed. To address these issues, tools may reserve routing tracks, or adjust routes, which can lead to congestion bottlenecks or delay violations. Thus, routing must account for signal integrity from the outset to avoid major redesigns.

In addition, the unpredictable behavior of electronic design automation (EDA) tools

also makes it challenging to achieve design convergence due to the cumulative impact of using complex heuristics used in different design stages. Small changes in the input or hyperparameter settings can cause large variations in the outcomes from EDA tools, making it hard to predict the quality of result (QoR) of the design, and complicating the design process.

To address the challenges discussed above, machine learning (ML) offers powerful approaches to improve the design efficiency and enhance the QoR. Unlike traditional methods, ML models can capture complex, nonlinear relationships between design parameters and post-layout outcomes, facilitate better modeling of downstream flow steps and provide proactive optimizations. Moreover, by integrating advanced ML models such as the reinforcement learning (RL) paradigm with traditional techniques, ML models can interact with EDA tools and to learn strategies for maximizing the long-term reward. For example, EDA tools offer a multitude of parameters that can be tuned to achieve optimal PPA outcomes. However, manual exploration of this configuration space is both time-consuming and resource-inefficient. Reinforcement learning has been applied to addresses this by training autonomous agents to iteratively learn and optimize tool settings without human intervention [4]. This, in turn, improves design efficiency and accelerates the search for optimal solutions, allowing design teams to maximize the potential of available resources.

1.2 Thesis contributions and organization

The first part of the thesis overviews the problem of timing estimation and timing optimization in late design stages. Initially, a more accurate delay prediction approach is proposed to address the inaccurate timing estimation issue in global routing [5]. Next, to resolve the timing failure in a late design stage caused by IR drop, a novel RL-based ECO gate sizing flow is proposed [6]. In the last part of the thesis, the problem of timing estimation for logic optimization during early design stages is studied [7]. The specific contributions of each work are summarized below.

ML-based parasitic and timing prediction for routing

Interconnect effects have been a critical factor in determining the performance of digital designs in modern technology nodes. As technology advances, increased resistance and capacitance have caused wire delays to become a major bottleneck in achieving design closure and overall IC performance. Accurate estimation of wire parasitics and delays is

essential to avoid unnecessary design iterations. Effective timing predictions can guide optimizations such as net buffering and logic gate resizing across multiple stages of the RTL-to-GDS flow. This work addresses the challenge of improving timing prediction accuracy in designs with macros during the global routing stage to enhance the quality of detailed routing solutions. The key contributions of this work includes (1) demonstrating how ML models enable fast and accurate predictions of parasitics and timing for detailed routing, leveraging global routing information, and (2) integrating ML models into the design flow in both open-source and commercial tool flows to improve design quality.

Reinforcement learning for ECO gate sizing

Power integrity tools and flows aim to restrict IR drops within specified limits, ensuring gate delay models are accurate for worst-case voltage corners. However, as designs approach final layout stages, wiring limitations often prevent the power grid from meeting IR drop constraints, leading to increased gate delays and potential timing failures. At this late stage, large design changes disrupt timing closure, necessitating incremental engineering change order (ECO) optimizations with minimal placement perturbation. To address this challenge, we propose a RL-driven framework for low-perturbation gate sizer to resolve IR-induced timing violations effectively. The key contributions of this work includes 1) combining RL and traditional approach support multi-objective optimization, 2) demonstrating better delay-power trade-off than traditional approaches and 3) supports diverse usage scenarios including zero-shot inference on unseen timing specifications or designs using a pretrained model and fine-tuning for improved performance.

ML-based timing prediction for logic optimization

The logic synthesis step plays a crucial role in determining the quality of final outcomes. The importance of efficient logic optimization has grown substantially as the technology node continues to scale down. The common drawback in many of previous methods is their reliance on proxy metrics as surrogates for delay and area. These proxy metrics often poorly correlate with actual post-synthesis delay after technology mapping, making them ineffective for optimizing modern designs. To address these challenges, more accurate ML inference for post-synthesis delay is integrated into the logic optimization flow in our work. The key contribution of this work include 1) correlation analysis of proxy metrics vs post-synthesis delay and proposing an efficient optimization flow, 2) propose an efficient ML-enhanced optimization flow and 3) demonstrating better solution exploration and delay-area trade-off.

The thesis is organized as follows. Chapter 2 provides an overview of a typical physical design flow and timing analysis at each design stage, and outlines the ML techniques that may be applied to the problems that we studied. Chapter 3 discusses the timing discrepancy between global routing and detailed routing in physical design and proposes ML models for more accurate parasitic and timing estimation. In Chapter 4, a novel ECO gate sizing flow for later design stages, using reinforcement learning, is proposed, using domain knowledge from prior EDA work to achieve high-quality solutions. Next, Chapter 5 discusses the poor correlation between the timing proxies used in logic optimization and post-synthesis timing and presents a new proposed logic optimization flow guided by ML-based timing estimation to for improving the quality of the design. Finally, Chapter 6 concludes the thesis.

Chapter 2

Preliminaries

Physical design is a phase in chip design that transforms a logic-level design, in the form of a *netlist*, into a physical representation, in the form of a *layout* for fabrication. This process involves several stages, each of which focuses on optimizing critical circuit metrics, i.e., the performance, power, and area (PPA). Design convergence involves some "chicken-and-egg" problems, where the accurate result or the optimization of one stage depends on the output of another stage, and vice versa. The dependency between layout optimization and the timing estimation is a key example: the layout must be optimized to improve timing metrics, but these timing metrics cannot be estimated unless wire parasitics are known, which can only happen after layout is complete.

In this chapter, we present a list of preliminary concepts that underpin the research presented in this thesis. Section 2.1 discusses an overview of layout and timing optimization, which form the backdrop of this research. Next, Section 2.2 overviews the key machine learning (ML) concepts that are used in the algorithms presented in this thesis.

2.1 Physical design and static timing analysis

Physical design takes a circuit netlist and creates a layout for the circuit. Fig. 2.1 shows a standard physical design flow. Since timing and physical design are intricately connected, the figure highlights multiple timing optimization stage that must be executed during physical design to repair design, i.e. fix max transition time violation and max fanout violation, and repair timing, i.e. fix setup and hold timing violation. Physical design starts from a gate-level circuit netlist generated from logic synthesis, and the

circuit is taken through mixed-size placement, clock tree synthesis (CTS), and routing. In mixed-size placement, both small standard cells and large macros are placed on the chip to optimize area, delay while avoiding overlaps and congestion. In CTS, the clock network is built to distribute the clock signal evenly with minimal skew and delay across all clocked elements. In routing, all components, i.e. cells and macros, are connected using metal wires, ensuring signal integrity and meeting timing, power, and design rule constraints.

It is essential to perform all timing optimizations in various parts of the flow in such a way that only limited netlist changes are introduced between successive stages. Limiting these changes is critical for ensuring a convergent design methodology, because a too-drastic netlist change between flow stages can force looping back to earlier steps of the flow, rather than continuing forward. However, due to the unavailability of routing information before the final stages, it is impossible to obtain accurate delay estimates at earlier steps of the physical design flow, and this could lead to incorrect optimizations at those stages.

2.1.1 Parasitic estimation for timing optimization

To overcome the challenge of delay estimation, design flows use different models to account for wire parasitics during timing optimizations, based on the information available at each stage. For example, as highlighted in Fig. 2.1, wireload models are used for gate-level optimization during logic synthesis. During global placement and even after CTS, generic half-perimeter wirelength (HPWL) or FLUTE-based [8] Steiner tree estimates, scaled by layer-averaged per-unit resistances and capacitances, are used to estimate parasitics for electrical rule check (ERC, including max load and max transition rules) compliance and some gate-level optimizations [9]. However, as we show in Section 3.1, these models can be highly inaccurate (overly conservative or grossly optimistic) compared to the true parasitics and timing estimates after detailed route. As a design proceeds from early stages (e.g., RTL specification, floorplanning) to later stages (e.g., detailed placement, detailed routing), an increasing amount of physical information (i.e., spatial embedding) is available, leading to potentially better estimates of parasitics.

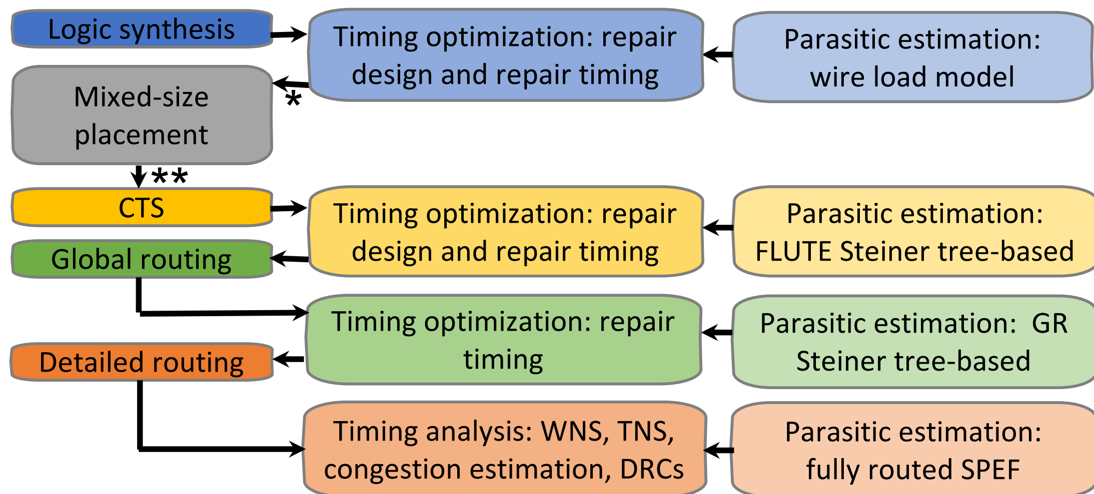


Figure 2.1: Timing optimizations in the physical design flow with different parasitic estimates. In modern production flows, the output of vendor A's synthesis tool is "de-buffered" when passed to vendor B's place-and-route (P&R) tool. Then, ERC compliance is enforced during global placement with buffering and resizing. The de-buffering and ERC-fixing steps are respectively marked with (*) and (**) in the figure. The wireload model is used to estimate parasitics for gate-level netlist. In placement and CTS stage, FLUTE Steiner tree is used. In global routing (GR) stage, GR Steiner tree that follows the route guide is used. After detailed routing, design rule checks (DRCs) are performed to ensures the layout adheres to the manufacturing process rules, and parasitic information are extracted into SPEF file. The worst negative slack (WNS) and total negative slack (TNS) are two metrics for timing analysis.

2.1.2 Static timing analysis (STA) and delay models

To perform timing optimization, it is first important to determine the timing behavior of a circuit. STA is fast approach used to validate the timing performance of a design without requiring simulation vectors. It ensures that the design meets setup and hold timing constraints across all specified conditions. For a given timing path, as illustrated in Fig. 2.2, the starting point may be either input port or the clock pin of a flip-flop, and the endpoint may be either the data input pin of a flip-flop or an output port. The data is launched by a clock edge at originating flip-flop, propagates through a combinational logic, and is captured by a clock edge at the destination flip-flop. To perform data setup checks, the worst-case path delay is calculated for each combinational block and compared against clock cycle constraints, with an allowance for setup time. For hold checks, the best-case path delays are compared against the hold time.

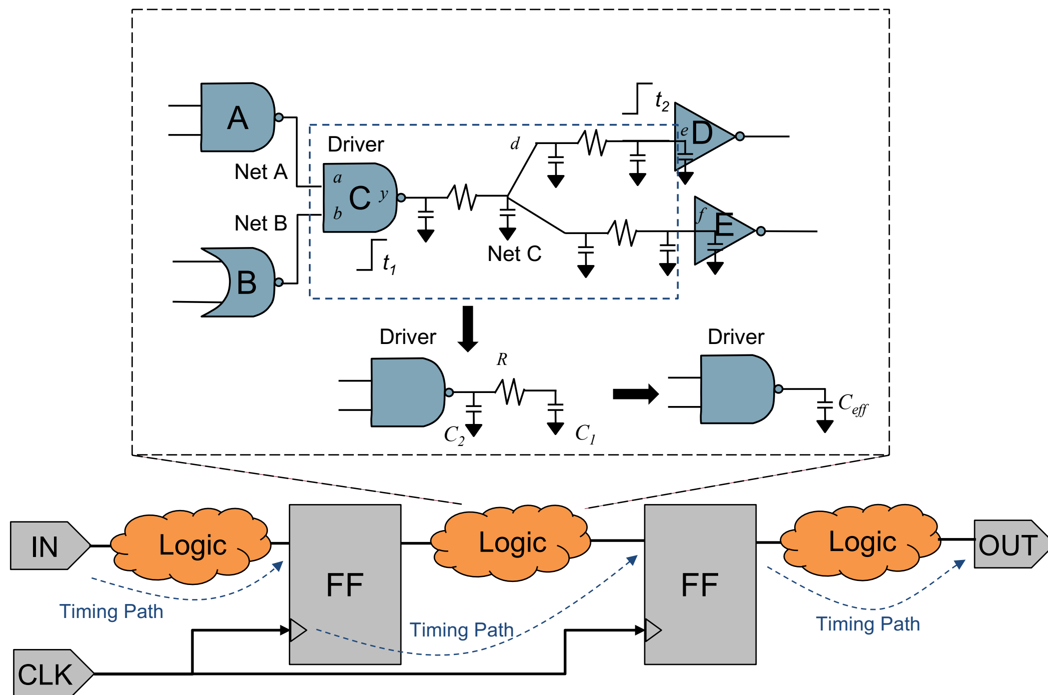


Figure 2.2: Timing paths in a design example and the RC tree model of a net in a logic stage, and its reduction to an equivalent π-model.

A unit operation in STA is the estimation of the delay of a single logic stage, consisting of a gate driving its fanouts through a net, as shown in the zoom-in part of Fig. 2.2.

The upper part of the figure shows the distributed RC tree for the net driven by gate C. The delay of a logic stage consists of the **gate delay** and **interconnect delay**. Given the estimated capacitive load C_{load} at the output of a gate (the "driving point"), and the transition time τ at the gate input, the gate delay is expressed through a lookup table (LUT) as

$$D_{gate} = f(\tau, C_{\text{load}}). \tag{2.1}$$

The gate delay for an intermediate pair of (τ, C_{load}) values that does not map to a LUT entry is computed using interpolation. The transition time at the gate output is estimated in a similar way, using LUTs that have the same axes as gate delay LUTs.

Since a wire is a distributed transmission line, it is modeled by the distributed RC model shown in Fig. 2.2, which segments the wire and creates lumped approximations for each segment. If the segments are small, this approximates the derivatives in the differential equation for a transmission line by finite sums. For a short wire, the wire resistance is dominated by the driver, and the wire can be represented by a capacitive load. Thus, the C_{load} model above can be used directly to compute the gate delay, which dominates the stage delay as the wire delay is negligible in this case. However, for longer interconnects, resistive shielding effects can significantly impact the delay [10]. For such wires, a segmented RC model cannot directly use the LUT of Eq. (2.1) as the load is not purely capacitive. A typical approach to overcome this in timing analysis is by (a) generating a π-model reduction for the driving point impedance at the gate output [11], with elements R, C_1, and C_2 (Fig. 2.2, bottom) chosen so that the first three admittance moments of the interconnect match those of the reduced model; (b) using the π-model to find an effective capacitance, C_{eff}, that models resistive shielding; and (c) using $C_{\text{load}} = C_{\text{eff}}$ in Eq. (2.1) to compute the gate delay. Finally, model order reduction techniques are used to compute the transfer function from the driving point to each fanout. Based on the delay and slew at the driving point, the waveform at each fanout is computed, yielding the wire delay and slew. The sum of the gate and wire delays constitutes the stage delay to the fanout, and the slew at each fanout is used as the input slew for the next logic stage.

2.2 Machine learning approaches

In this section, we overview a set of ML techniques that are used in this thesis. We first describe the ensemble decision tree models and several kinds of graph neural networks (GNNs) that are often used for classification and QoR prediction. Next, we describes the reinforcement learning (RL) paradigm used to trains an agent that iteratively learns to optimize the PPA of the design.

2.2.1 Ensemble decision tree models

Decision trees are classical ML models used for classification and regression tasks. A decision tree makes decisions by recursively splitting the data based on feature values, resulting in a tree structure where each leaf node represents the predicted outcome. When datasets become complex, the simple decision tree models are prone to overfitting. Thus, ensemble models are proposed to combine multiple decision tree base learners to improve prediction accuracy and robustness. By evaluating predictions from a collection of models, ensemble models achieve better generalization and reduce the overfitting. Two widely used ensemble methods are Random Forest and XGBoost. Random forest is a bagging-based model that takes the average of the prediction from multiple decision trees or makes the decision by majority voting [12]. Each decision tree is trained on a subset of the data and features in parallel. XGBoost is a boosting-based model that build decision trees sequentially to correct the prediction error sequentially [13].

2.2.2 Graph neural networks

GNNs are deep learning architectures designed to operate on graph-structured data. GNNs are particularly useful for representing circuit structures where nodes can correspond to gates, and edges capture connectivity. There are several widely used models: graph convolutional networks (GCNs), graph attention networks (GATs), graph sample and aggregate (GraphSAGE).

GCNs leverage spectral methods to propagate information across graph nodes, aggregating features from neighboring nodes through learnable weights [2]. The propagation rule for a single GCN layer is given as

$$H(l+1) = \sigma\left(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}H(l)W(l)\right) \tag{2.2}$$

where $H(l+1)$ is the feature matrix at hidden layer $l+1$, $W(l)$ is the weight matrix

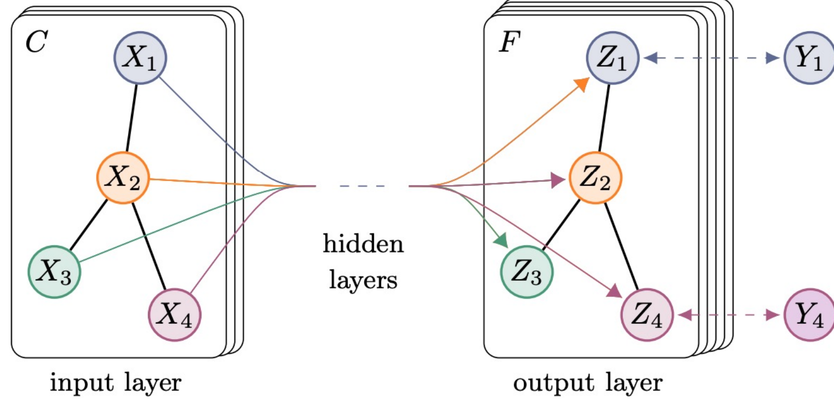


Figure 2.3: Multi-layer GCN with C input channels and F feature maps in the output layer. X_is is the node features vectors of node i at the input layer, and Z_i is the node feature vector of node i at the output layer. The graph structure is shared over layers, where edges are shown as black lines. The labels of nodes are denoted by Y_i [2].

at hidden layer l, $\tilde{A}$ is the adjacency matrix with added self-loop, $\tilde{D}$ is the degree matrix and σ is the non-linear activation function. GCNs have been used for graph classification, graph embedding and prediction of the quality of the design in physical design.

Relational Graph Convolutional Networks (R-GCNs) extend GCNs by modeling heterogeneous graphs that include multiple types of nodes and edges [14]. This flexibility allows R-GCNs to handle complex relationships, making them relevant for tasks involving heterogeneous graphs with diverse node type and different relation between nodes in circuit design. The propagation for R-GCN is given as

$$h_i(l+1) = \sigma\left(W_o(l)h_i(l) + \sum_{r\in\mathcal{R}}\sum_{j\in\mathcal{N}_i^r}\frac{1}{d_{i,r}}W_r(l)h_j(l)\right) \tag{2.3}$$

where $h_i(l+1)$ is the feature vector for node i at layer l, $W_o(l)$ is the trainable weight matrix for self-connections, $W_r(l)$ is the trainable weight matrix for edge type r, $\mathcal{N}_i^r$ is the set of neighbors of i under relation r, and $d_{i,r}$ is the normalizing constant for instance i that have the relation r.

GATs incorporate an attention mechanism into GNNs [15], allowing the network to assign varying levels of importance to neighbors when aggregating node features. The node representation is aggregated as

$$h_i' = \sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{ij} W h_j\right) \tag{2.4}$$

where α_{ij} is the attention coefficient normalized by softmax function defined as

$$\alpha_{ij} = \frac{\exp(\text{LeakyReLU}\left(a^T\left[Wh_i \| Wh_j\right]\right))}{\sum_{k\in\mathcal{N}_i}\exp(\text{LeakyReLU}\left(a^T\left[Wh_i \| Wh_k\right]\right))} \tag{2.5}$$

The learnable attention vector a^T multiplies with the concatenation of the dot products of the weight matrix W and the embeddings of two nodes i and j, and then a LeakyReLU activation function is applied.

GraphSAGE is a framework for inductive representation learning on graphs. It's designed to generate node embeddings for unseen data by learning aggregation functions from sampled neighborhoods [16]. Unlike transductive models such as GCN, which require access to the entire graph during training, GraphSAGE learns a function that generalizes to new nodes or graphs, making it effective for dynamic and large-scale graphs.

2.2.3 Reinforcement learning methods

RL is a machine learning paradigm where an agent learns to make sequential decisions by interacting with an environment. It is a powerful tool for tackling the complex optimization problems in modern chip design. An overview of the RL paradigm is shown in Fig. 2.4.

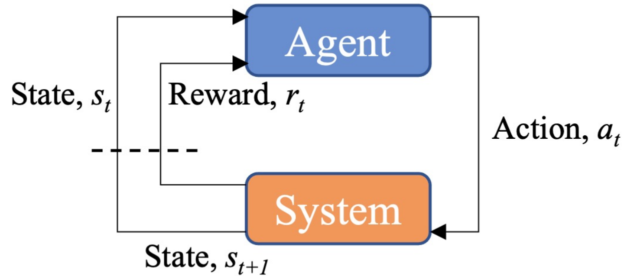


Figure 2.4: Schematic of the RL paradigm, showing interactions between an RL agent and a system.

The key components of an RL system include

- State space S, a set of states which describe the environment.
- Action space A, a set of actions that the RL agent can take.
- $T(s_{t+1}|s_t, a_t)$, transition dynamics that map a state s_t with action a_t at time t to state s_{t+1} at time $t+1$.
- $r_t = r(s_t, a_t, s_{t+1})$, the immediate reward function.
- $\gamma \in [0, 1]$, a discount factor. A lower value places a larger emphasis on immediate reward, while a higher value places greater emphasis on future reward.

In the RL set-up, the agent is at a state s_t at timestep t. In this state, the agent takes an action $a_t \in A$, the state transitions from s_t to s_{t+1}, with an immediate reward r_t at time step t. When we concatenate all time steps together, we obtain a sequence of transitions $(s_0, a_0, r_0, s_1) \longrightarrow (s_1, a_1, r_1, a_2) \longrightarrow ...$, which is called an episode that fully describes the trajectory of the agent. The agent aims to maximize a cumulative reward by choosing actions based on its current state. Unlike supervised learning, RL does not require labeled data but instead relies on feedback to guide its learning process. The goal of RL is to find an optimal policy $\pi(s)$, a strategy that an agent follows in pursuit of the goal, to maximize the expected reward G_t, which is the cumulative discounted reward:

$$G_t = \sum_{t=0}^{\infty} \gamma^t r_t \tag{2.6}$$

The state-value function $V_\pi(s)$ is the expected return when the agent starts in state s and its actions follow the policy $\pi(s)$. It can be computed recursively according to the Bellman Equation [17]:

$$\begin{aligned} V_\pi(s) &= E[G_t|S_t = s] \\ &= R(s, \pi(s)) + \gamma \sum_{s'} P(s, \pi(s), s') V_\pi(s') \end{aligned} \tag{2.7}$$

where $R(s, \pi(s))$ is the reward of a transition from state s through action determined by $\pi(s)$, S_t is the observed state at time step t, P is transition probability between state s and state s', $P(s, \pi(s), s') = Pr(S_{t+1} = s'|S_t = s, A_t = \pi(s))$, and s' is a neighboring state of the current state s.

A quality function $Q_\pi(s, a)$ can be constructed similarly for an action a taken in

state s under the policy $\pi(s)$:

$$\begin{aligned} Q_\pi(s,a) &= E[G_t|S_t = s, a_t = a] \\ &= R(s,a) + \gamma \sum_{s'} P(s,a,s')V_\pi(s') \\ &= R(s,a) + \gamma \sum_{s'} P(s,a,s')E_{a'\sim\pi}Q_\pi(s',a') \end{aligned} \tag{2.8}$$

Here, $V_\pi(s')$ is replaced by $E_{a'\sim\pi}Q_\pi(s',a')$, which is the expected value of a given state s' is the expectation of the estimated values over all possible actions at state s'.

RL algorithms are classified as on-policy algorithms and off-policy algorithms. The on-policy algorithms learn from actions taken by the current policy, improving the same policy that is used to generate data. Off-policy algorithms learn from actions taken by a different policy, often using stored experiences or exploratory policies, enabling reuse of past data. The on-policy algorithms are more stable but they require more data since they cannot reuse past experiences. The off-policy algorithms are sample-efficient and better suited for tasks requiring data reuse but may face stability challenges due to the mismatch between the behavior and target policies.

In RL algorithm selection for chip design problems, the action space is one of the primary factors to consider. When the action space is discrete, deep Q-learning network (DQN) [18], A2C [19] and their variants are often considered such as Rainbow [20], a variant of DQN which combines six other DQN variants into a single model to achieve better performance, and A2C with experience replay [21] that combines multiple networks used in A2C and the replay buffer used to stores the trajectories of experience in DQN to improve the stability of the off-policy estimator. When the action space is continuous, proximal policy optimization (PPO) [22] that uses a ratio of the new and the old polices to limit the update step size, deep deterministic policy gradient (DDPG) [23] that combines DQN and determinant policy gradient, soft actor-critic (SAC) [24] that uses entropy regularization related to the randomness in the policy to maximize the trade-off between the exploration and exploitation, are popular choices.

There are several advantages of applying RL in chip design. One of the key benefits is its ability to learn from dynamic and complex environments to optimize solutions for problems with large and combinatorial search spaces. Additionally, RL models can continuously refine their policies to deliver better PPA for designs. However, RL also has some drawbacks. RL often requires a large number of interactions with the

environment to learn effective policies, which is sample inefficient in real chip design scenarios. Also, RL models are often considered black-box solutions, which make them lack of interpretability.

Chapter 3

Bridging the timing Gap between Global Route and Detailed Route Using ML

3.1 Introduction

Interconnect effects are a significant factor in determining the post-layout performance of digital designs in modern integrated circuit technology nodes. In particular, due to increased per-unit length resistances and capacitances from one technology generation to the next, wire delays have become a significant bottleneck in achieving design closure and overall IC performance outcomes. The accurate estimation and prediction of wire parasitics and delays is particularly important in modern design flows, as incorrect estimates can result in unnecessary design iterations; in some cases, it may be impossible to achieve design closure on schedule, which may result in increased time to market. If wire parasitics and delays can be estimated well, they can be used to guide timing optimizations such as net buffering and logic gate resizing at multiple stages of the RTL-to-GDS implementation flow.

Even though at relatively late stages in the physical design flow, such as routing, there can be significant estimation inaccuracy. As shown in Fig. 2.1, a design is typically routed in two stages: global routing (GR) and detailed routing (DR). The GR step allocates routing resources to each net, and generates a routing plan that the DR step takes as initial guidance toward a final routing solution. In modern tools, the routing

plan from GR is represented in the form of *route guides* that contain information on layer assignments and Steiner tree topologies for each net [25] [26], and these are used to guide DR. Clearly, the ability to close timing depends on the quality of the GR route guides.

Importantly, the GR stage is typically followed by a timing optimization step (sizing, fanout clustering, buffering, etc.) as well as a final (post-optimization) placement legalization step, since better parasitic estimates are available post-GR compared to after earlier flow stages. However, GR-based parasitic estimates are still inaccurate relative to final DR outcomes, as they do not fully comprehend such factors as detailed design rules, pin access challenges, and congestion, which are glossed over in GR. Together, these factors cause wire detours and layer reassignments during DR, which in turn cause GR-based estimates and DR-based outcomes to diverge.

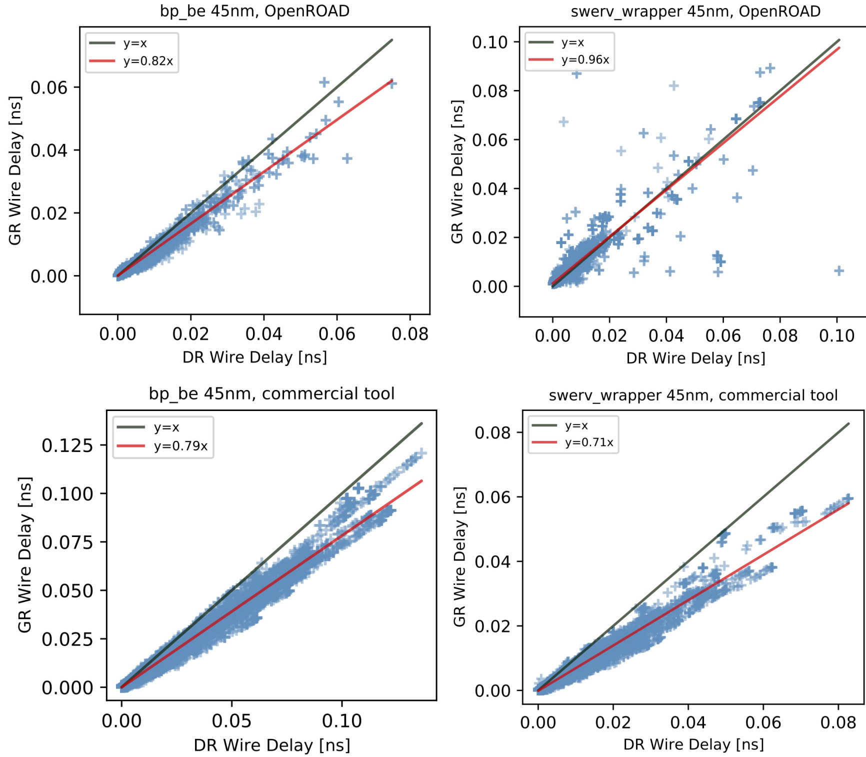


Figure 3.1: Discrepancy between post-GR and post-DR wire delays of bp_be 45nm and swerv_wrapper 45nm in OpenROAD (left figures) and bp_be 45nm and swerv_wrapper 45nm in a commercial tool flow (right figures). The slope k and the intercept b in the best-fit $y = kx + b$ (red traces) differ across tool flows and across designs.

Fig. 3.1 shows the inaccuracy in wire delay estimation when using a GR-based model that estimates wire parasitics using FLUTE-generated [8] Steiner trees from FastRoute 4.1 [27], as compared to the wire delay of corresponding detail-routed nets when using RC trees from post-DR parasitic extraction. The figures show discrepancies in the estimated wire delay of four designs: bp_be (42K nets) and swerv_wrapper (88K nets), each implemented in 45nm technology in both OpenROAD and commercial tool flows. Each plot shows a $y = x$ line: a perfect prediction would lie along this line; an optimistic prediction, where the GR-based delay estimate is smaller than the DR-based ground-truth delay, would lie below the line; and a pessimistic prediction would lie above the line. In OpenROAD, the GR-stage estimates are largely optimistic for bp_be, but may be either pessimistic or optimistic for swerv_wrapper. For the commercial tool flow, the GR-based wire delay estimates for the most part are optimistic. While it could be argued that the prediction can be corrected by using a best linear fit line ($y = kx+b$) instead of the $y = x$ line, this does not repair all of the discrepancies. Specifically, we draw the following conclusions:

- The slope k for calibrating the best-fit curve is tool-flow-specific, and we see different values for the OpenROAD flow and the commercial tool flow.
- Even for the same tool flow, the best-fit curve is design-specific: the value of k for bp_be is significantly different from that for swerv_wrapper. This is observed to be true for both the OpenROAD and commercial tool flows.
- Compared to the discrepancy observed in OpenROAD, the wire delay discrepancy for the commercial tool flow is lower, reflecting a superior ability to adhere to GR route guides. Even so, the discrepancy is significant.

Therefore, the problem of predicting post-DR interconnect delays from post-GR route guides is more complex than a simple correction through a curve-fit, and this motivates our approach of using a machine learning (ML) predictor.

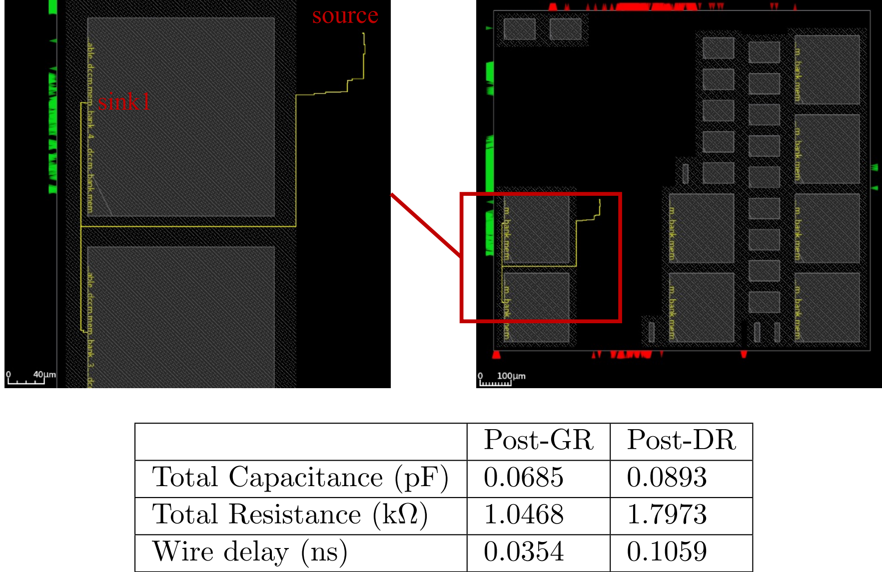
"Oracle" knowledge of post-DR parasitics improves final outcomes. As explained earlier, the use of inaccurate parasitic and timing estimates in any timing optimization can lead to harmful pessimism (overdesign that wastes resources) or optimism (underdesign that leads to design iterations). We have conducted a motivating study to show the potential benefit of "oracle" knowledge of post-DR wire parasitics, were these parasitics to somehow be available to post-GR timing optimizations. Table 3.1 shows results on four open-source 130nm [28] testcases, seven open-source 45nm [29] testcases

Table 3.1: Comparison of post-DR WS when using GR-based vs. DR-based parasitics for post-GR timing optimizations.

Design	**CLK (ns)**	**Tech**	**#Nets**	**#Macros**	**Utilization**	**Post-DR WS (ns)**	
						GR-based parasitics	DR-based parasitics
riscv32i	9.6	130nm	8150	0	67.52%	-0.26	-0.26
aes	5.4		15307	0	62.91%	-0.21	-0.19
ibex	16.0		15369	0	55.85%	-0.56	-0.60
jpeg	7.8		59573	0	62.90%	-0.25	-0.17
dynamic_node	1.0	45nm	11598	0	57.63%	-0.25	-0.17
ibex	2.0		16836	0	58.94%	-0.24	-0.11
aes	0.8		17566	0	59.60%	-0.27	-0.09
jpeg	1.4		68247	0	52.48%	0.24	0.02
swerv_wrapper	2.5		88490	28	45.87%	-0.24	-0.23
bp_fe	2.2		24883	11	43.73%	-0.15	-0.10
bp_be	2.8		41973	10	38.30%	-0.15	-0.12
swerv_wrapper	1.2	12nm	92787	28	49.08%	-0.48	-0.24
coyote	3.2		272948	15	49.00%	-0.27	-0.15

and two commercial 12nm testcases, using an open-source flow [30]. As indicated in the table, some of these layouts consist of standard cells only, with no macros, while others contain a mix of standard cells and large macro blocks. The table highlights the cost of post-GR buffering and resizing solutions that are driven by inaccurate parasitics, and also lists the utilization for each design. For example, if the discrepancy as in Fig. 3.1 is corrected in post-GR for each design, the improvement in the post-DR worst slack (WS) for these designs can be as much as 240ps (-0.48ns $\rightarrow$ -0.24ns). We ascribe the WS improvement to the early identification of true (post-DR) timing violations, which allows post-GR optimizations to efficiently buffer nets and resize logic gates on truly critical timing paths. In the absence of accurate prediction, these timing paths might be missed due to optimism, or unnecessarily buffered and resized due to pessimism. This motivates the key result of our work, which is to apply ML to close the post-GR-to-post-DR parasitic estimation gap. On closer inspection, we see that the designs with smaller post-DR WS improvement in Table 3.1 correspond to high values of utilization. For example, riscv32i has the highest utilization among 130nm designs without macros, and swerv_wrapper has the highest utilization among 45nm designs with macros. In these designs, fewer options are available for resizing and buffering in post-GR timing optimization, resulting in fewer perturbations due to these operations. This may partially explain why post-DR worst negative slack (WNS) from the flows based on GR-based

and DR-based parasitics are quite close.



	Post-GR	Post-DR
Total Capacitance (pF)	0.0685	0.0893
Total Resistance (kΩ)	1.0468	1.7973
Wire delay (ns)	0.0354	0.1059

Figure 3.2: Net detours due to macro blockage in swerv_wrapper, 45nm. Total capacitance, total resistance and wire delay of source-sink1 in the sample net.

While significant inaccuracies are seen even for designs without macros, the presence of large macros in a design is a source of another major degree of difficulty in delay prediction. Depending on their structure, these macros may act as partial or complete blockages for interconnect wires, depending on whether some or all metal layers are blocked by the macro. The wiring detours that are required to circumvent the blockages can greatly impact the routing solution – notably, the wirelength, wire delay, and the need for buffer insertion along these wires. As an example, Fig. 3.2 shows a placement of swerv_wrapper in 45nm with several macros. The figure at right shows the overall layout, with a region of interest shown within a red rectangle, with two large macros placed close to each other. At left, we zoom into this rectangle, highlighting a three-pin net that connects a source pin at one side of two macros to two sinks at the other side of the macros: clearly this routing solution is required to detour around these macro blockages. From source to sink1, the wire bypasses a macro of size $206.9\mu m \times 219.8\mu m$ through the halo between two macros, and the source-sink1 wire is 30% longer than the Manhattan distance between the two pins. The impact of the detour on key performance metrics for the net are listed in the table at the bottom of the figure: its post-DR capacitance and

resistance are, respectively, 22.5% and 71.7% higher than those predicted at the end of GR, and its wire delay discrepancy between the post-GR and post-DR steps is 199.2%. Therefore, a critical characteristic of any prediction model must be its capability to predict interconnect delays for designs with macros.

Another practical consideration is related to the fact that EDA tools are inherently noisy [31], and can provide different results for nearly-identical inputs and runscripts. Therefore, the noisy data collected from these tools can affect the estimation of final results and may lead to incorrect buffering and sizing during optimization, thus affecting design quality. For example, a small perturbation in the location of an instance can result in a different route. This is illustrated in Fig. 3.3, where a buffer is inserted between a NAND gate and an OR gate during timing optimization to reduce the delay of the NAND gate. However, depending on the precise location of the inserted buffer – two possibilities are shown in Fig. 3.3(a) and Fig. 3.3(b) – the delay between the NAND gate and OR gate may be different due to dissimilarities in the wire delay. Therefore, the capability of estimation models to handle noise in the training data is another critical evaluation criterion. Due to the high cost of generating training data, it is not practical to explore the entire "noise space" around a particular training point; instead, we verify that the ML-based models we build are resilient to such noise.

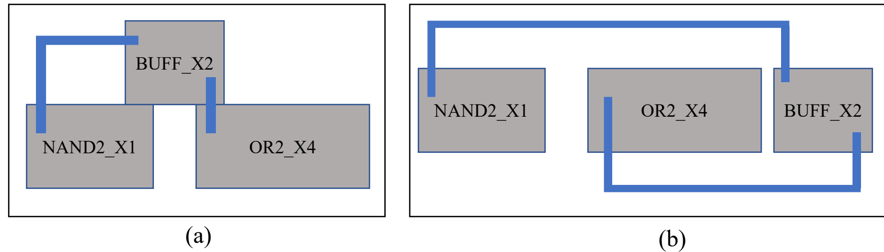


Figure 3.3: The effect of different buffer locations on the routing results: (a) The inserted buffer is placed in a different row of the NAND gate and OR gate. (b) The inserted buffer is placed in the same row as the NAND gate and OR gate.

Related works. Several researchers have worked in the general area of ML-based delay prediction during physical design. For a specified net topology, [32] builds an XGBoost-based ML model for the wire delay of a net of fixed topology, trained on commercial parasitic extraction and timing analysis tools. The impact of macro block layout on timing is predicted using boosting and SVM in [33]. The work in [34] solves the problem of predicting wire delay and slew based on placement results, prior to GR,

while [35] predicts path delays prior to routing, based on placement features, using a transformer network and residual model. The work in [36] uses a look-ahead RC network generated by a coarse routing step (decomposition of multi-pin nets into two-pin nets, then routed using L-shaped routes) on the placed design for feature extraction, and uses this to perform post-placement net-based timing prediction. In [37], a GNN model is used to estimate prerouting slacks at the endpoints of the design. In [38], wire delay and wire slew are predicted by using a graph learning architecture that encodes the information of the RC tree of a net.

Contributions. In this work, we propose a set of techniques to enhance the accuracy of timing prediction on designs with macros in the post-GR stage to improve the quality of the DR routing solution. The key contributions of this work are as follows:

1. We show how ML enables the fast and accurate prediction of post-DR parasitics and timing estimates using post-GR information, particularly for design with macros, in both bulk and FinFET technology nodes.

2. We apply the ML model to the OpenROAD [39] physical design flow and to a commercial flow, and show up to 0.24ns savings (12nm node) in post-DR WS for OpenROAD, and 0.32ns savings (45nm node) for the commercial tool flow, without degrading congestion.

3. We demonstrate that a similar flow that uses ML models for timing estimation can also be applied within a commercial tool flow, operating within the constraints of the information available through the available APIs, to improve the timing prediction accuracy and DR solution quality.

4. We find that as compared to a traditional flow, with a small increase in runtime (to perform ML inference), the ML model can improve the mean %error of path slack from 5.75% to 1.15% in a 45nm testcase, and from 14.91% to 7.61% in a 12nm testcase.

5. The proposed timing prediction models are assessed to be generalizable with respect to designs generated with different clock constraints, and can handle small noise in datasets.

3.2 Preliminaries

3.2.1 Post-GR and post-DR routing estimates

Timing-driven physical design requires an estimate of the delay of each stage of logic. This corresponds to the sum of the gate delay and the wire delay, each of which is dependent on wiring parasitics. To predict circuit timing, post-GR interconnect parasitics may be estimated based on the route guides. The estimated RCs for a net are determined by the length and the topology of the GR-constructed Steiner tree used for GR and the layer assignments. In this work, we use FastRoute [27], which performs precise layer assignment for each route, i.e., the route guides specify the precise layer for each segment of the Steiner tree.

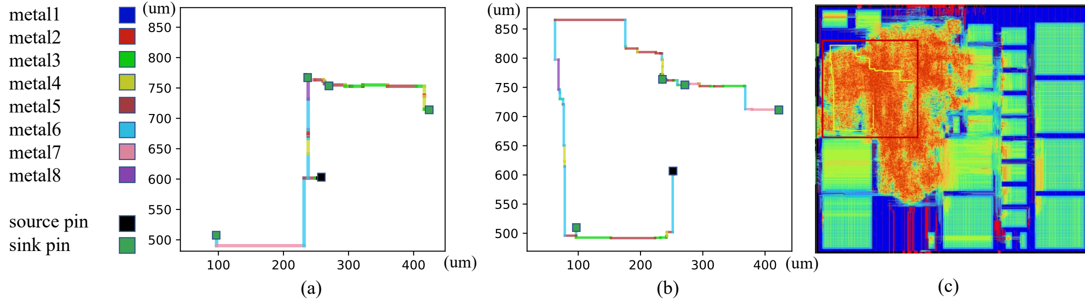


Figure 3.4: Differences between the GR guides and post-DR routes of a 5-pin net from swerv_wrapper 45nm. (a) Route guides in post-GR. (b) Post-DR routes. (c) Net highlighted in congestion map.

The precise wire routes and wire adjacency relations are available for all nets only after DR is complete. Therefore, post-DR parasitic extraction provides the exact ground-truth parasitics for each route. For a 45nm swerv_wrapper layout, the GR and DR solutions for a 5-pin net that lies in a high-congestion region are displayed in Figs. 3.4(a) and (b), respectively. Fig. 3.4(c) superposes the net over the post-DR congestion map for the design: the red box shows the region displayed in Figs. 3.4(a) and (b), and the net is highlighted in yellow. It is clear that the DR solution of this net chooses a path that is significantly different from that specified in GR, and that this is due to limited available routing resources in the congested region: compared to the GR guides, the DR solution makes a large detour to either bypass the most congested region or overcome pin access issues. Due to such differences between the post-GR wire route prediction

and post-DR wiring topologies of the nets, the timing results based on post-GR parasitics and the ground truth can sometimes be quite different. As a result, GR-stage timing estimates can be inaccurate. Moreover, depending on factors such as changes in the Steiner tree, or the coupling capacitances to neighboring wires, post-GR parasitic estimation may be pessimistic or optimistic, as seen in Fig. 3.1. This may mislead post-GR timing optimization steps (e.g., buffering/resizing) into performing incorrect or inaccurate optimizations.

3.3 DR timing prediction framework

3.3.1 Timing prediction in routing flow

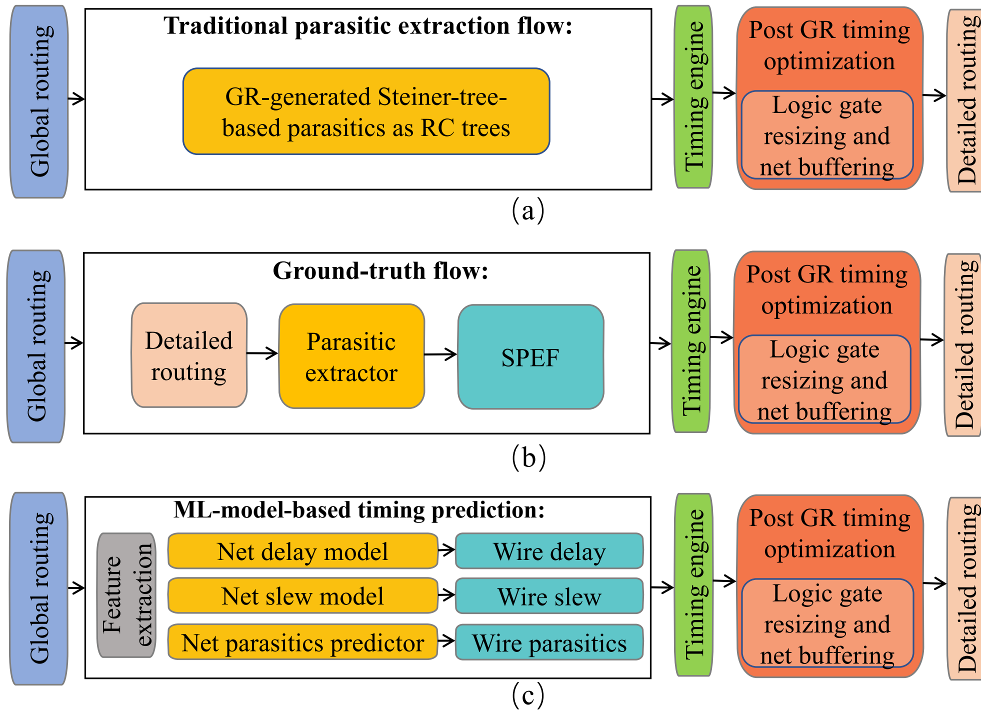


Figure 3.5: Flows that use different parasitic estimates for post-GR timing optimizations: (a) traditional flow (Steiner tree-based RC estimates), (b) ground-truth (DR followed by parasitic extraction to determine post-DR parasitics), and (c) our flow (fast ML engine for post-DR parasitics and timing prediction).

Fig. 3.5(a) highlights a typical routing flow in physical design, where the GR stage

is followed by timing optimization before the final DR stage. These optimizations rely on parasitic estimates from route guides, which use Steiner trees to construct an RC tree network. This results in timing inaccuracy, relative to the final DR timing (see Fig. 3.1). In an ideal flow, shown in Fig. 3.5(b), performing DR and extracting parasitics would provide accurate timing estimates, but such a flow is impractical due to the high computational expense of DR.

We propose the use of an ML-based flow to predict post-DR timing from features extracted at the post-GR stage in both OpenROAD and a commercial tool flow, as highlighted in Fig. 3.5(c). Through fast ML inference, the model can rapidly predict post-DR timing estimates without performing time-intensive DR. Our framework leverages three XGBoost-based ML models to predict the following three post-DR metrics:
(i) Source-sink wire delay: Our ML model is applied on a per-sink basis, and it predicts the delay between the driving point and the sink pins. For example, in Fig. 2.2, for net C, the model predicts the wire delay between the driving point (pin y of the driver gate C) and the sink (pin e of gate D). Similarly, it predicts the source-sink delay between the driver pin y and sink pin f.
(ii) Source-sink wire slew: Our ML model is used to predict post-DR wire slew at each sink in the design. We predict source-to-sink wire slews, i.e., the difference in the transition times between the waveforms at the driver pin and at each of the corresponding sink pins. In Fig. 2.2, the source-sink wire slew is $(t_1 - t_2)$, where t_1 is the transition time at the driving point pin y, and t_2 is the transition time at the sink pin e.
(iii) Wire parasitics: In the OpenROAD flow, our ML model predicts post-DR π-model parasitics (R, C_1, and C_2, as shown in Fig. 2.2). The timing engine uses these three parameters to estimate C_{eff} which is used, in turn, to calculate gate delays. For the commercial flow, the model predicts the post-DR total load capacitance. In the OpenROAD flow, the predicted wire delay, wire slew and π-model parameters are annotated by corresponding commands through Tcl APIs, and the STA engine updates C_{eff} and gate delays based on the π-model parameters. In the commercial tool flow, equivalent commands are used to annotate the predicted wire delay and wire slew, and the parasitics are annotated. However, the available APIs do not expose the π-model parameters, and therefore, we settle for using the total load capacitance, C_{load}, for these designs. For reasons described in Section 3.5, this does not result in significant inaccuracies in the commercial tool flow.

Together, the above ML models (i)-(iii) estimate post-DR circuit delays with the

help of an STA engine for annotated delay propagation. The first two models are directly used to annotate wire delays and wire slews in the timer while the latter is used indirectly in the timer to calculate C_{eff}, which is used as C_{load} in (2.1) to compute the gate delay.

3.3.2 ML engine

All three ML models are implemented using XGBoost [13]. As shown in Section 2.2, it is an ensemble learning algorithm based on gradient boosting. XGBoost predicts the target variable using parallel tree boosting, combining estimates from several models including gradient-boosted decision trees.

3.3.3 Feature engineering

Input features for source-sink delay and slew prediction models

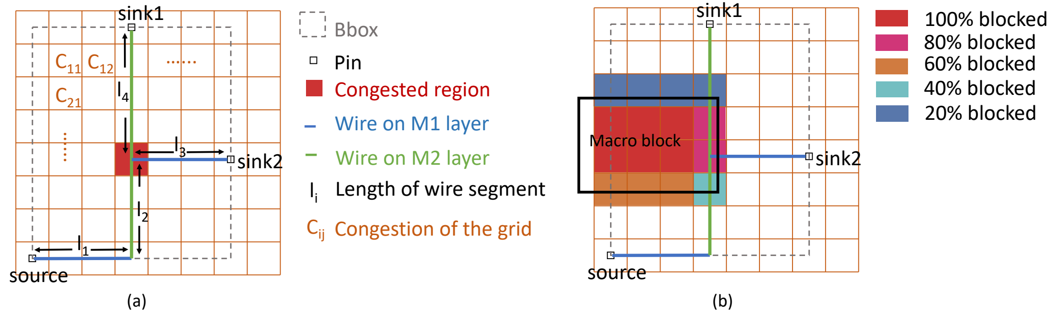


Figure 3.6: A route guide for a three-pin net with a source and two sinks. (a) The net is routed using four wire segments with lengths l_1, on M1; l_2, on M2; l_3, on M1; and l_4, on M2. (b) The net traverses two GCells that are 80% blocked by a macro and two GCells that are 40% blocked and 20% blocked.

We identify the following set of input features that are used to build the ML model that predicts the source-sink delay and slew of a net. The first set of features described below are derived on a per-net basis, while another set of features, described later, are on a per-sink basis.

HPWL: This feature (half-perimeter of the minimum bounding box of the net) is illustrated by the dashed bounding box in Fig. 3.6(a). It provides a lower bound on the wirelength.

Number of sinks: It is well known that for nets with numerous sinks, the HPWL can be a significant underestimate. Multi-sink nets show larger discrepancies between post-GR and post-DR wirelengths as they tend to detour and/or have net lengths that exceed the HPWL, as noted above. We therefore pass this feature, which is extracted from the gate-level netlist, to the ML model.

Slew at the driving point: The driving point slew indicates the signal strength at the source pin. This slew is extracted from a timing analysis tool, which uses GR-generated Steiner tree-based parasitics. The slew at the source pin affects both source-sink wire delay and the wire slew: a smaller driving point slew leads to a smaller source-sink delay and slew.

Congestion estimates: As illustrated in Fig. 3.4, congestion is critical to bridging the discrepancy between GR and DR. Nets whose GR-generated route guides go through congested regions tend to detour in DR, for routability, pin access, and DRC-related reasons. Therefore, congestion estimates are essential for predicting wire detours. As features, we use both the mean and standard deviation of the congestion in all grid cells (GCells) in the bounding box of the net as shown in Fig. 3.6(a).

Rise and fall transitions: Since the wire delays and slews are different for the rise and fall transitions, we encode the switching direction with a binary-encoded feature (0 for rise and 1 for fall).

The above-listed features are on a per-net basis, i.e., identical for all sinks on the net. Since we predict wire delays and slews on a per-sink basis, we also use the following sink-specific features:

Source-sink length: The source-sink length is defined as the total length of the wire segments that connect the driving point to the target sink pin, and is extracted from the GR-generated route guides. For the example net in Fig. 3.6(a), routed in layers M1 and M2, the source-sink length for sink1 is defined as $(l_1 + l_2 + l_4)$. Since the source-sink length is strongly reflected in to the source-sink delay and slews, this is a critical feature in estimating post-DR wire and slew delays.

Source-sink R, C: These features give the total resistance and capacitance, respectively, between the driving point and the target sink. The total resistance [capacitance] is the sum of the products of the segment length and the per-unit resistance R_{Mi} [capacitance C_{Mi}] of its assigned layer Mi, over all source-to-sink wire segments. In Fig. 3.6(a), the total resistance to sink1 is $(l_1 \times R_{M1} + l_2 \times R_{M2} + l_4 \times R_{M2})$, and the total capacitance is $(l_1 \times C_{M1} + l_2 \times C_{M2} + l_4 \times C_{M2})$.

Routing blockage due to macros: Since macro blocks squeeze the space available

for routing, any nets traversing the GCells blocked by macros must detour around these GCells, with probability that is a function of the degree of blockage; the detouring resutls in increased wire delay, total wire resistance, and total wire capacitance. As shown in Fig. 3.6(b), the path from the source to sink1 traverses four GCells that are blocked by macros: two of these GCells are 80% blocked, one is 40% blocked, and one is 20% blocked. Due to limited routing resources, it is likely that the wire connecting source and sink1 will bypass the GCells that are 80% blocked in the final routing result. The features associated with the macros must capture the variation in GCell blockage over the regions traversed by a net. This may be achieved in several ways, e.g., (1) using the mean of the percentage blockage of the GCells in the bounding box of an entire net; (2) using the maximum percentage blockage of the GCells in a source-to-sink bounding box; or (3) using the mean of the percentage blockage of the GCells in the bounding box for a source-sink pair.

However, for a net with multiple sinks, the first option using the mean blocking percentage of the GCells in the net bounding box can be misleading: it does not estimate the macro impact accurately when one source-sink path goes through a highly blocked GCell (e.g., >80% blocked) but other source-sink paths traverse less blocked or unblocked GCells. The second option, using the maximum blocking percentage of the GCells in the source-sink bounding box, could mispredict the wire detour when only a small number of GCells are largely blocked by macros. Therefore, we base our approach on the third option, using the mean blocking percentage of all GCells in the source-sink bounding box, to estimate the impact of macro blockage on the wiring inside the source-sink bounding box.

Furthermore, we observe that even this approach is imperfect. Notably, a detoured route may traverse GCells outside of the bounding box, and the macro blockages in these GCells outside the bounding box will also affect the routing result. Hence, the average blocking percentage in an expanded bounding box is introduced as a feature to estimate the macro impact, where the expanded bounding box is scaled by scaling factor $\alpha > 1$ over the source-sink bounding box. Empirically, we find that $\alpha = 2$ is a good choice and we therefore use an expanded bounding box with $\alpha = 2$, i.e., doubling the length and width of the bounding box, to determine the mean percentage blockage that is used as a feature in the ML model.

Input features for the load prediction model

For the ML model that predicts the parameters of the π-model at the driving point for use with the OpenROAD timer, we use the HPWL, number of sinks, congestion estimates and macro blockage estimates as features. We use additional features related to the values of R, C_1, and C_2, generated by applying the O'Brien/Savarino model [11] to the GR-generated Steiner tree. Since the timing engine of the commercial tool uses C_{load} instead of C_{eff}, and does not provide visibility into the parameters R, C_1, and C_2 of the π-model, our best option is to use HPWL, number of sinks, congestion estimates and macro blockage estimates, along with C_{load} based on the GR result, as model features for the commercial tool flow.

3.4 Model training and inference in the physical design flow

3.4.1 Ground-truth data generation and model training

Our ground-truth data is generated using the flow highlighted in Fig. 3.5(b). Since our experiments are performed in different tool flows, we use both a branch of OpenROAD-flow-scripts [30] and a commercial flow. The ground-truth data is generated for multiple designs implemented in two open-source bulk CMOS technologies (a 45nm technology [29]) and a 130nm technology [28], and one commercial FinFET technology. All of the tested designs are summarized in Table 3.2.

To increase the diversity of our training data set, we run the ground-truth flow for each design with three different floorplan areas and placement utilization settings. Different utilization settings result in designs with different levels of congestion and consequently in nets that have different lengths due to placements and detours. We extract the features described in Section 3.3.3 and their corresponding ground-truth labels. The training labels for the three ML models are extracted after timing analysis (corresponding to the green box in Fig. 3.5(b)) including the source-sink wire delays, source-sink wire slews, and load parameters. Each source-sink pair represents a single data point for the source-sink delay and slew at the sink. For the load capacitance predictor, each net in the design is a single data point. The training data is normalized before training; however, in practice we observe that the XGBoost model is not very sensitive to this normalization and performs adequately using skewed data. The models

Table 3.2: Summary of designs implemented in 45nm, 12nm, and 130nm technology nodes.

Design	Tech	# Nets	Macros
ibex	45nm	17566	0
aes		16836	0
jpeg		68247	0
dynamic_node		11598	0
swerv_wrapper		88490	28
bp_fe		24883	11
bp_be		41973	10
swerv_wrapper	12nm	92787	28
coyote		272948	15
ibex	130nm	15307	0
aes		15369	0
jpeg		59573	0
riscv32i		8150	0

for designs with macro blocks and without macro blocks are trained separately. For designs without macros, our ground-truth dataset contains 456,661 data points in 45nm technology, 348,429 data points in 12nm technology, and 394,162 data points in 130nm technology. For designs with macros, our ground-truth dataset contains 880,225 data points in 45nm, and 643,793 data points in 12nm. The pace of ground-truth data generation is slow (1 hour per design, on average) but this is a one-time cost per technology since the trained model can be applied to new designs to rapidly and accurately predict post-DR parasitics and timing.

The model is trained using root mean squared error (RMSE) as the loss function. For the XGBoost regressor [13], we use a learning rate 0.01. We choose the maximum tree depth = 4, the number of estimators = 900, and the subsampling ratio = 0.8. The hyperparameter values are chosen based on a full search of a discrete grid on the domain of the hyperparameters: the assignment with the highest score is used for parameter tuning. We find that the models are not sensitive to small changes in the hyperparameter values.

3.4.2 ML inference in physical design flow

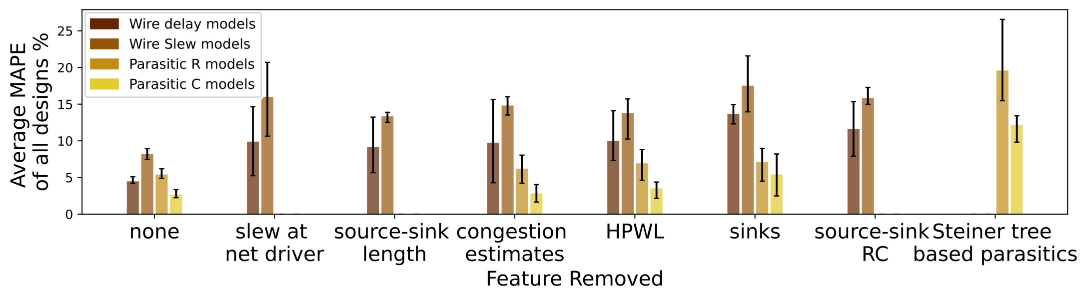
The trained ML models are applied to the flow shown in Fig. 3.5(c). At the post-GR phase, the ML model features are extracted from the design data and route guides.

Then, the features are fed into the three ML models, which perform a fast and accurate inference to predict source-to-sink wire delays and wire slews, as well as π-model parameters in OpenROAD or effective load capacitance in the commercial tool flow. The predicted estimates are annotated via Tcl APIs in the STA engine: load parameters are used by the timing engine to estimate the gate delay, while the source-sink wire delays and slews are directly used as the net delays and net transition times. The timer performs an update to propagate these annotated wire delays and gate delay estimates. The new ML-predicted timing estimates are used by the timing optimizer to perform gate resizing and buffer insertions which fix setup, hold, maximum slew, maximum fanout, and maximum load violations.

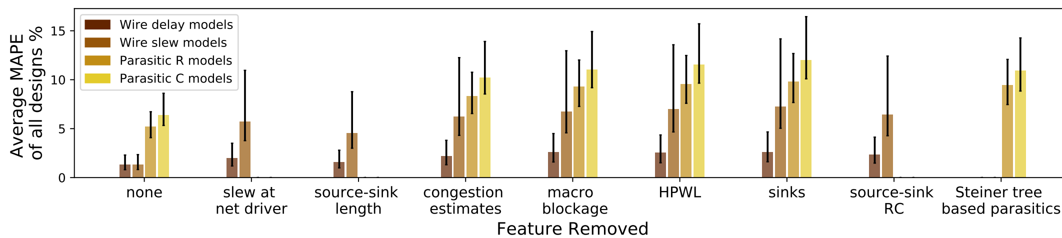
As confirmed in our experimental studies (Seciton 3.5 below), the ML-based inference flow provides an accurate estimate of post-DR timing without performing time-intensive DR. These estimates are useful for efficient buffering and resizing, as the optimizer now has improved knowledge of the truly (post-DR) critical paths.

3.4.3 Feature sensitivity

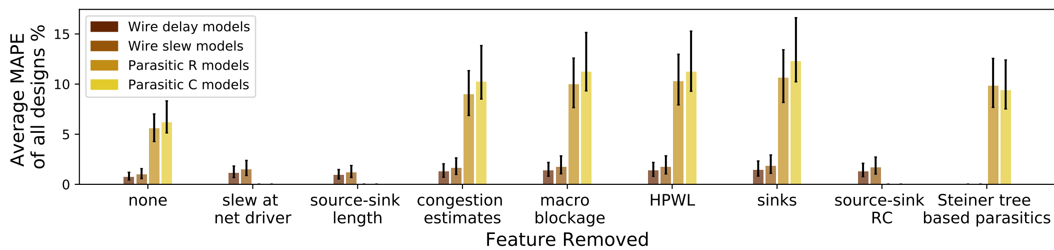
To demonstrate that each selected feature is indispensable to the model, we perform a sensitivity analysis for each feature, for each of the four models (wire delay, wire slew, parasitic R, and parasitic C) from Section 3.3.1. Note that not all models use all features; therefore, there are missing bars for certain features in the figure. Fig. 3.7 performs an ablation study on the set of features, removing one feature at a time and examining its impact on the accuracy of the ML model for designs with macros. The analysis of the ML model for designs without macros was conducted in our previous conference version [40]. For example, for source-sink wire delay prediction, we measure the test accuracy for wire delay prediction models, each trained by removing one specific feature at a time. Similar experiments are also performed on the wire slew model, parasitic R model, and parasitic C model. The y-axes of Figs. 3.7(a)-(c) respectively show the average of the mean average percentage errors (MAPEs) over all 45nm designs without macros, all 45nm designs with macros, and all 12nm designs with macros. The x-axis of each figure lists the feature that has been removed. The figure is annotated with an error bar that shows the maximum and minimum %error across all the designs. We find that the model has the best test accuracy when all the features are selected. Thus, each feature contributes to improving the accuracy of the model. For the 12nm model, which has lower parasitics, the errors approximately double with removal of slew and



(a) Average MAPE on designs without macros in 45nm.

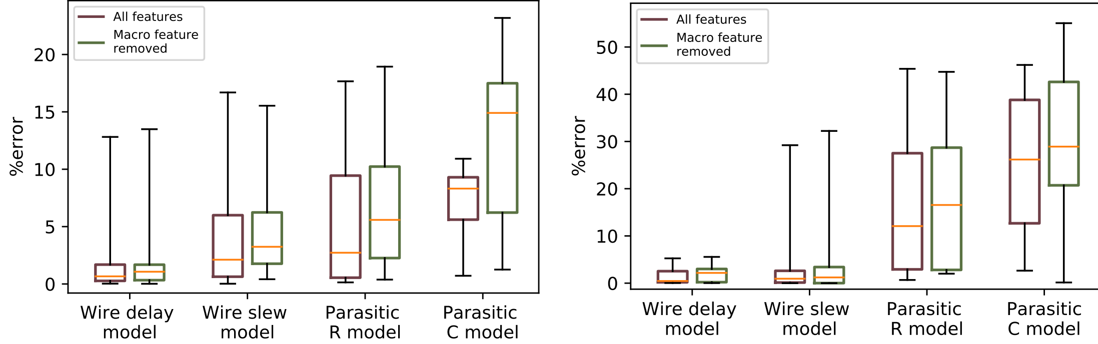


(b) Average MAPE on designs with macros in 45nm.



(c) Average MAPE on designs with macros in 12nm.

Figure 3.7: Feature sensitivity analysis showing the average mean absolute percentage error (MAPE) (y-axis), for each removed feature (x-axis), for all 45nm designs and all 12nm designs.



(a) Macro feature sensitivity for 45nm designs. (b) Macro feature sensitivity for 12nm designs.

Figure 3.8: Macro feature sensitivity for wires that have area in a bounding box that with >80% blockage.

source-sink features, but since the baseline ("None") is low, the error after removal is also low. Nevertheless, we keep these features in the model to enable generality.

We further examine the macro feature sensitivity in Fig. 3.8 for the four ML models. The x-axis of the figure lists the models trained with the macro features, and the y-axis highlights the average percentage error of the nets in all 45nm designs and all 12nm designs. Nets whose expanded bounding box has a large overlap with macro blockages are much more greatly affected by macro features than those that are more distant. For these nets, we plot data for source-sink pairs that have 80% of their source-sink bounding box area blocked by macros. Fig. 3.8 shows that introducing macro features improves the average absolute %error from (1.87%, 3.24%, 5.58%, 14.90%) to (1.41%, 2.11%, 2.72%, 8.31%) for 45nm designs, where the four percentages correspond to average absolute %error of the four models in the figure. For 12nm designs, the corresponding numbers in the average absolute %error show an improvement from (2.17%, 1.21%, 16.56%, 28.90%) to (0.45%, 0.94%, 12.07%, 26.17%).

3.5 Experimental setup and evaluation

Our experiments are performed on benchmarks from OpenROAD across three technologies: two open technologies, NanGate 45nm [29] and SkyWater 130nm [28], and one 12nm commercial technology. We perform physical design using two flows: with OpenROAD and a commercial EDA tool flow. Our target designs fall into two classes: those without macros (IBEX, AES, JPEG and Dynamic Node in 45nm; IBEX, AES,

JPEG and RISCV32I in 130nm; and IBEX, AES and JPEG in 12nm) and those with macros (swerv_wrapper, bp_fe and bp_be in 45nm; and swerv_wrapper and coyote in 12nm). As compared to the preliminary version of this work [40], where the results were shown only on bulk nodes, we now show results here on a FinFET technology node; we apply the approach using not only OpenROAD, but also using a commercial tool flow; and we expand the algorithm to address designs with macros. In addition, we test the ability of our models to generalize with respect to clock periods by running inference on unseen designs generated by using different clock constraints, and we also test the robustness of our models to noise by adding Gaussian noise to training datasets. As detailed below, these comprehensive experiments confirm the general applicability of our approach.

The ML models for designs with and without macros are trained separately. In Table 3.3, we compare the prediction results of *one shared* model for both designs with and without macros, versus the results of *two separate* models for designs with macros and without macros. For the training of the shared model, we leave out each test design from all designs in 45nm and use the remaining designs for training. The table shows that the model trained specifically for designs with macros or designs without macros has better accuracy than a shared model trained for all designs. We explain this by observing that macro-induced detours and blockages pose specific challenges due to the 100% blockage and the large sizes of the macros, which cannot be easily captured in the shared model. Therefore, building separate models is better than using a shared model. For designs without macros, the models tested on each design are trained using data from the remaining designs without macros in the same technology. For designs with macros, we are handicapped by the limited number of available designs: i.e., only three designs in the 45nm node and two in the 12nm node. Therefore, the models are trained by using the data from other designs with macros at various utilization settings, and also from the same design, but for different utilization settings. Thus, even with this limited dataset, we ensure that the ML model is evaluated for an unseen design having a different utilization than the set of designs in the training set.

Training and inference for the ML model are implemented in Python 3.6 and performed on a machine with Intel Xeon Silver 4214 CPU @2.2GHz and NVIDIA A100 PCIe 40GB GPU. For the OpenROAD flow, the predicted parasitics, wire delays, and wire slews are annotated into the timing engine by modifying the OpenROAD [41] and OpenSTA [42] source code. For the commercial flow, the predicted values are annotated into the timing engine through available Tcl commands in the commercial tool flow.

Table 3.3: Prediction error of the shared ML model, and the separate ML models for designs with macros and designs without macros in 45nm.

Designs	**Clock period**	**# Macros**	**Wire delay**						**Wire slew**					
			ML model (separate)			**ML Model (shared)**			**ML model (separate)**			**ML Model (shared)**		
			Mean	Max	Stdev.	Mean	Max	Stdev.	Mean	Max	Stdev.	Mean	Max	Stdev.
swerv_wrapper	2.5ns	28	0.84%	115.19%	2.78%	2.95%	149.41%	4.83%	0.04%	19.81%	0.34%	0.55%	22.72%	0.61%
bp_fe	2.2ns	11	0.64%	19.14%	0.91%	2.76%	32.34%	2.78%	0.04%	4.42%	0.13%	0.47%	6.46%	0.65%
bp_be	2.8ns	10	2.16%	34.58%	3.42%	2.93%	48.90%	3.06%	0.07%	9.15%	0.25%	0.49%	8.85%	0.59%
dynamic_node	1.0ns	0	4.21%	39.82%	5.00%	6.60%	53.29%	7.58%	8.19%	39.96%	8.18%	8.96%	67.78%	7.60%
ibex	2.0ns	0	4.30%	38.38%	5.99%	4.93%	35.33%	3.54%	8.94%	38.32%	7.35%	9.51%	50.48%	7.06%
aes	0.8ns	0	5.12%	39.90%	7.46%	7.35%	52.59%	10.43%	8.23%	39.98%	8.17%	14.88%	65.03%	12.62%
jpeg	1.4ns	0	4.13%	39.68%	5.56%	8.41%	54.71%	8.01%	7.47%	39.97%	9.09%	10.54%	46.30%	8.12%

We evaluate our ML-based flow against the traditional ("Trad.") and ground-truth-based flows for (i) accuracy, and (ii) impact on post-DR outcomes – worst slack (WS), total negative slack (TNS), runtime and congestion. To highlight the importance of incorporating macro-based features, we also compare the results from the new flow, using a macro-based ML model, to a flow similar to our preliminary work [40], where the models are trained without macro features.

3.5.1 Model accuracy evaluation

We analyze the accuracy of the ML models at both the net/sink level (the ML model performs inference on a per-net/per-sink basis) and at the path level. It is critical to evaluate the predicted timing at both levels, as net-level errors have the potential to accumulate or cancel during delay propagation.

Net-level accuracy

Table 3.4 summarizes the metrics for the ML models and the accuracy for 45nm designs and 12nm designs in both OpenROAD and the commercial tool flow, and 130nm designs in OpenROAD, evaluated on designs with and without macros. We use the mean, maximum, and standard deviation of the absolute %error as metrics for evaluation, where the absolute percentage error is the absolute difference between the ground-truth and the predicted value with respect to a reference. The references that we use are the stage delay for evaluating wire delay; the ground-truth sink slew for wire slew; and the ground-truth labels for parasitics.

The table shows better mean %error and maximum %error metrics for our ML-based wire delay and wire slew predictions compared to GR Steiner-tree-based estimation from both OpenROAD and the commercial tool flow for most designs. In general, the ML-based model shows significant reductions in the %error. In only two cases, denoted

Table 3.4: Evaluation of the ML models using mean, maximum, and standard deviation of the absolute %error as metrics.

Tool	Designs	Clock period	Technology node	Wire delay						Wire slew						Path delay					
				ML-based			GR-based			ML-based			GR-based			ML-based			GR-based		
				Mean	Max	Stdev.	Mean	Max	Stdev.	Mean	Max	Stdev.	Mean	Max	Stdev.	Mean	Max	Stdev.	Mean	Max	Stdev.
OpenROAD	swerv_wrapper*	2.5ns	45nm	0.84%	115.19%	2.78%	4.27%	149.41%	7.28%	0.04%	19.81%	0.34%	0.20%	40.63%	1.22%	0.34%	4.62%	0.30%	1.48%	7.54%	0.69%
	bp_fe*	2.2ns		0.64%	19.14%	0.91%	3.63%	15.68%	4.65%	0.04%	4.42%	0.13%	0.24%	5.83%	0.98%	0.65%	1.91%	0.41%	0.69%	2.47%	0.52%
	bp_be*	2.8ns		2.16%	34.58%	3.42%	3.00%	55.64%	3.88%	0.07%	9.15%	0.25%	0.12%	10.45%	0.38%	0.63%	2.21%	0.45%	0.80%	2.57%	0.34%
	dynamic_node	1.0ns		4.21%	39.82%	5.00%	8.43%	51.33%	6.45%	8.19%	39.96%	8.18%	10.56%	51.51%	10.54%	0.57%	5.84%	0.32%	2.35%	10.76%	2.43%
	ibex	2.0ns		4.30%	38.38%	5.99%	6.54%	49.47%	9.62%	8.94%	38.32%	7.35%	11.52%	49.40%	9.47%	0.70%	6.64%	0.42%	2.11%	10.66%	1.18%
	aes	0.8ns		5.12%	39.90%	7.46%	11.60%	68.43%	9.62%	8.23%	39.98%	8.17%	10.61%	54.53%	10.53%	0.82%	4.21%	0.30%	2.63%	11.04%	1.72%
	jpeg	1.4ns		4.13%	39.68%	5.56%	5.32%	57.15%	7.17%	7.47%	39.97%	9.09%	9.63%	51.52%	11.72%	2.82%	10.60%	2.03%	6.07%	29.78%	4.28%
	swerv_wrapper*	1.2ns	12nm	0.87%	22.80%	0.93%	4.26%	44.95%	5.76%	0.04%	29.81%	0.17%	0.70%	71.75%	3.85%	1.15%	7.61%	1.26%	5.75%	14.91%	4.77%
	coyote*	3.2ns		0.60%	26.58%	0.88%	3.47%	39.65%	4.73%	0.22%	40.13%	0.62%	1.97%	51.34%	4.51%	1.09%	2.65%	0.27%	1.23%	7.73%	1.51%
	ibex	16.0ns	130nm	3.35%	21.21%	6.33%	4.32%	27.34%	8.16%	4.15%	21.54%	5.45%	5.35%	36.77%	7.03%	0.64%	8.17%	1.05%	1.09%	4.99%	1.13%
	aes	5.4ns		11.15%	39.05%	8.03%	25.37%	72.34%	10.35%	2.46%	29.18%	3.15%	3.17%	65.61%	4.06%	0.47%	3.22%	0.40%	1.48%	7.77%	0.53%
	jpeg	7.8ns		4.17%	37.83%	6.51%	14.38%	48.76%	8.39%	5.84%	39.97%	6.22%	7.53%	67.52%	8.02%	0.82%	6.87%	0.65%	3.20%	12.65%	2.31%
	riscv32i	9.6ns		1.54%	3.44%	0.67%	2.99%	8.43%	0.86%	1.15%	2.58%	0.83%	1.48%	3.33%	0.84%	1.08%	2.85%	0.42%	4.13%	19.90%	3.27%
Commercial tool	swerv_wrapper*	2.2ns	45nm	7.21%	63.59%	6.50%	29.80%	70.56%	11.56%	0.24%	9.14%	0.71%	1.46%	50.93%	5.28%	0.35%	5.49%	0.53%	1.00%	5.94%	0.60%
	bp_fe*	1.6ns		5.88%	43.79%	4.96%	24.41%	51.07%	9.26%	0.36%	6.08%	0.79%	3.73%	37.84%	8.83%	0.54%	4.81%	0.46%	1.64%	7.05%	0.67%
	bp_be*	2.0ns		6.09%	57.99%	5.46%	25.84%	56.00%	9.29%	0.36%	19.30%	0.83%	3.04%	99.94%	7.84%	0.60%	5.08%	0.47%	1.65%	6.28%	0.50%
	swerv_wrapper*	1.0ns	12nm	6.24%	36.47%	1.27%	12.30%	56.30%	2.78%	0.36%	26.26%	0.95%	0.85%	79.14%	2.78%	3.52%	8.28%	2.29%	5.98%	13.69%	3.24%
	coyote*	3.2ns		5.22%	83.62%	1.05%	15.42%	98.70%	2.99%	3.68%	59.35%	0.88%	11.75%	93.02%	2.86%	2.22%	7.36%	1.74%	4.67%	10.85%	2.87%

in red, the error for our approach is worse than the GR-based approach, but we have verified that these errors are from short nets that have negligible wire delay and small stage delay, and therefore do not affect the critical path in the circuit.

In some cases in the table, the maximum %error is very large (e.g., over 100% for swerv_wrapper 45nm). We have confirmed that this error can also be attributed to a short wire that also has a small stage delay as a reference. Scatter plots of the wire delay and wire slew comparisons in ns, without normalization, are presented in Fig. 3.9, showing a more complete picture of the match between ML prediction and the ground truth, relative to the GR-based approach. It can be seen that the match for the ML predictor is considerably better than for the GR-based approach (even the outlier at the right of the second plot is significantly closer to the $x = y$ line than for the GR-based case). Since the large errors are only in cases where the ground-truth wire delay and wire slew are very small, the path delay is not greatly affected by these errors. From the table and the scatter plots, it can be seen that the standard deviation of %error from ML models is also lower, indicating that few nets have larger errors. For our application of post-GR timing optimization, these accuracy levels are sufficient to realize the benefit of the ML models.

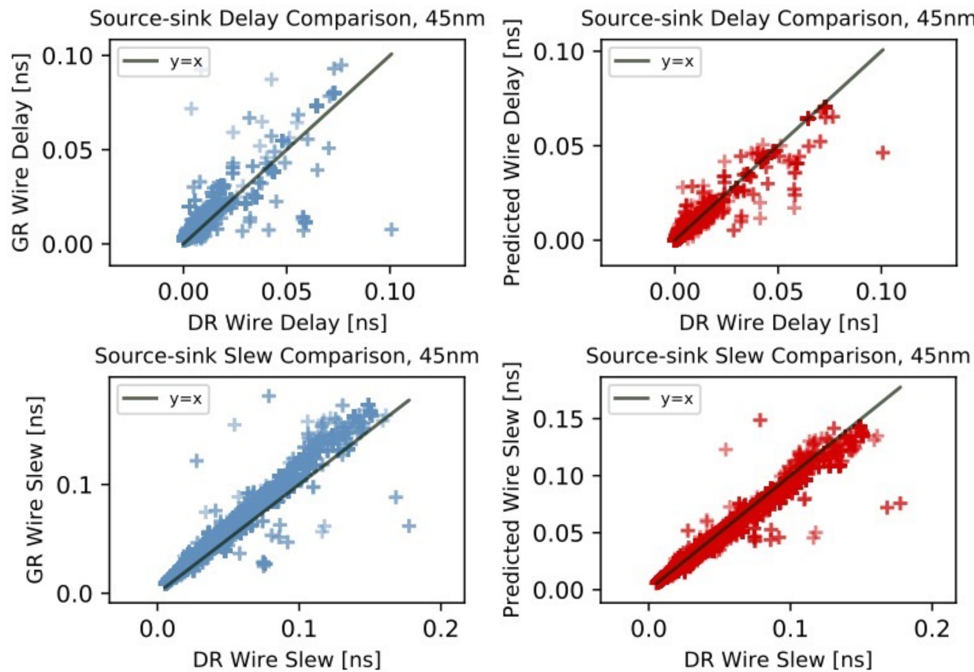


Figure 3.9: Wire delay and slew comparisons for swerv_wrapper 45nm.

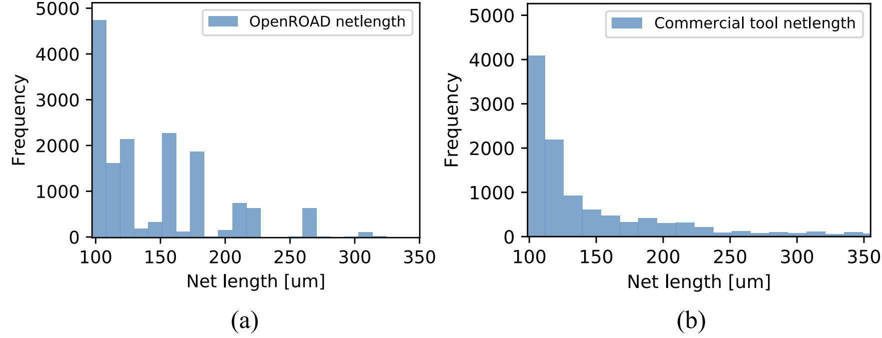


Figure 3.10: Wirelength distribution of swerv_wrapper 45nm: (a) for a design with size of $1.20 \times 1.09mm^2$ generated in OpenROAD, and (b) for a design with size of $1.18 \times 1.18mm^2$ generated in a commercial tool flow.

It is worth noting that the accuracy of the commercial-tool-flow-based flows is generally lower than that of the OpenROAD-based flow. The reason why OpenROAD provides better accuracy is because we have visibility into the full π-model, while the commercial tool API only provides a view of the total C_{load}. However, the error in the commercial flow is still acceptable. To examine this further, we show histograms of the distribution of wirelengths for a representative design, swerv_wrapper in 45nm, under both flows. Fig. 3.10 shows the number of nets with lengths greater than 100μm under both flows. From the histograms, it is apparent that the designs generated by the commercial flow have fewer long wires as shown in Fig. 3.10; this can be attributed to lower utilizations for OpenROAD *vis a vis* the commercial tool flow. Due to these shorter wirelengths, the simpler C_{load} feature is adequate to capture the wire capacitance, and is supplemented by the source-to-sink wirelength feature that acts as a surrogate for a wire resistance feature. For the OpenROAD-based flow, with longer wires, we find that the use of these features results in significant accuracy loss, and that the π-model features are required to achieve the accuracy levels shown in Table 3.4.

Path-level accuracy

We analyze the accuracy of the predictor in estimating the delays of a sample of paths in the circuit. These path delays are estimated by annotating the predicted parasitic values, wire delays, and wire slews into a timing engine. We compare the path slacks across the traditional flow and the ML-based flow. Specifically, we analyze paths whose slack is below 40% of the clock period, considering one worst-case path for each endpoint.

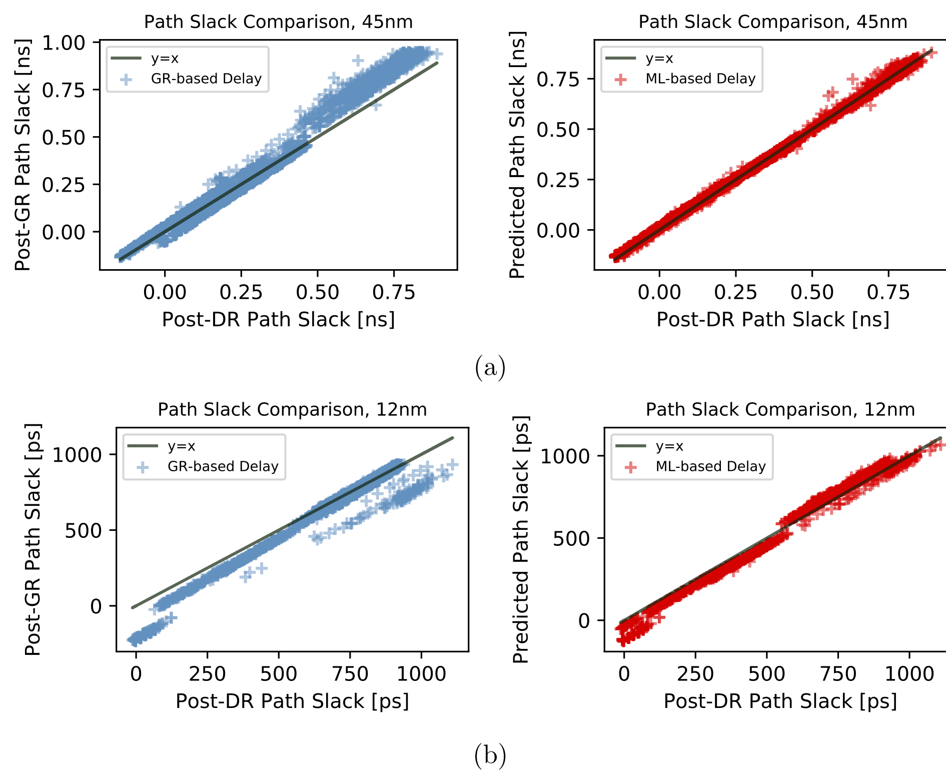


Figure 3.11: Path slack comparison for (a) swerv_wrapper 45nm and (b) coyote 12nm.

Table 3.5: Impact of ML-based prediction for designs with macros on post-DR metrics for traditional, ground-truth and ML-based flows in OpenROAD.

Design	**Tech**	**Die size (mm^2)**	**Utilization**	**Post-DR WS (ns)**				**Post-DR TNS (ns)**				**Runtimes (s)**		
				Trad.	**ML based**	[40]	**Ground Truth**	**Trad.**	**ML based**	[40]	**Ground Truth**	**Trad.**	**ML based**	**Ground Truth**
swerv_wrapper CLK=2.5ns	45nm	1.20x1.09	45.87%	-0.14	-0.14	-0.13	-0.13	-23.45	**-21.96**	-22.78	-22.58	1629	1657	1995
		1.36x1.20	39.22%	-0.20	-0.20	-0.20	-0.20	-38.40	-37.22	-37.69	-35.03	1601	1630	1881
		1.55x1.34	30.71%	-0.24	-0.23	-0.23	-0.21	-57.22	**-43.22**	-50.16	-49.16	1588	1613	1898
bp_fe CLK=2.2ns		0.78x0.63	43.73%	-0.15	-0.10	-0.13	-0.10	-1.60	-0.95	-1.40	-0.72	700	707	891
		0.84x0.68	37.45%	0.01	0.02	0.01	0.02	0.00	0.00	0.00	0.00	619	628	681
		0.99x0.79	27.09%	-0.01	**0.02**	-0.01	0.00	-0.01	0.00	-0.03	0.00	598	604	709
bp_be CLK=2.8ns		0.75x0.75	41.82%	-0.18	-0.17	-0.17	-0.15	-9.63	-7.82	-7.92	-6.60	792	798	1217
		0.79x0.78	38.30%	-0.15	-0.12	-0.12	-0.10	-10.86	-6.81	-7.76	-4.15	768	774	988
		0.98x0.92	25.95%	-0.09	**-0.07**	-0.09	-0.10	-2.30	-2.10	-2.63	-2.09	622	629	889
swerv_wrapper CLK=1.2ns	12nm	0.64x0.48	49.08%	-0.48	**-0.24**	-0.32	-0.36	-207.68	-202.39	-204.37	-200.55	6041	6060	13861
		0.70x0.54	39.86%	-0.35	**-0.23**	-0.27	-0.33	-364.02	-343.81	-347.05	-337.56	5418	5433	12072
		0.80x0.64	29.39%	-0.21	-0.18	-0.20	-0.18	-306.18	**-234.98**	-284.44	-237.94	4846	4865	10900
coyote CLK=3.2ns		0.66x0.66	49.08%	-0.02	0.05	0.04	0.09	-0.14	0.00	0.00	0.00	7243	7251	13405
		0.75x0.75	35.87%	-0.27	-0.15	-0.19	-0.14	-24.73	**-12.93**	-20.50	-14.06	6029	6038	10883
		0.85x0.85	27.91%	-0.08	-0.05	-0.06	-0.04	-1.21	-0.94	-1.04	-0.75	6054	6064	11465

Fig. 3.11 shows an example of the slack comparison for (a) swerv_wrapper in 45nm technology and (b) coyote in 12nm technology. The figures on the left show the discrepancy in path slack between GR and DR timing estimates and the figures on the right show the ML-corrected path slacks versus the post-DR path slacks. With the ML-based timing correction applied after GR, the post-GR path slacks have a better match with post-DR slacks.

The last six columns in Table 3.4 compare the mean, maximum and standard deviation of path delay %errors from the ML model against the %errors from the traditional flow for multiple designs in both OpenROAD and the commercial tool flow. The mean path delay %error is defined as the mean of the absolute percentage difference between the ML-based delays and the post-DR path delays, using the clock constraint for each design as the (normalization) reference. The traditional flow has a higher mean, maximum, and standard deviation of %error when compared to the ML-based flow, with just one exception – the standard deviation of bp_be 45nm in OpenROAD is slightly worse than with the traditional flow. This indicates that on average the ML-based post-GR delays correlate better with post-DR delays than the traditional-flow-based delays. The path delay accuracy is improved by our approach, with the average error reducing from 2.50% to 1.11%. This enables timing optimizations to buffer nets and resize logic gates on truly critical paths.

3.5.2 Impact on post-DR outcomes

The three ML models are applied to the timing analysis step before post-GR timing optimization transforms, based on sizing and buffering, in the physical design flow. We

Table 3.6: Impact of ML-based prediction on post-DR metrics for traditional, ground-truth and ML-based flows in a commercial tool.

Design	Tech	Die size (mm^2)	Utilization	Post-DR WS (ns)			Post-DR TNS (ns)			Runtime(s)		
				Trad.	ML based	Ground Truth	Trad.	ML based	Ground Truth	Trad.	ML based	Ground Truth
swerv_wrapper CLK=2.2ns	45nm	1.18x1.18	48.85%	-0.04	-0.02	-0.01	-0.22	-0.12	-0.01	629	661	2435
		1.25x1.25	40.76%	0.00	0.00	0.00	0.00	0.00	0.00	501	533	1995
		1.44x1.44	36.91%	-0.08	-0.08	-0.04	-0.89	-0.52	-0.40	507	602	2243
bp_fe CLK=1.6ns		0.63x0.63	50.72%	-0.29	-0.24	-0.01	-10.96	-8.75	0.00	257	265	867
		0.71x0.71	41.77%	-0.31	-0.15	-0.04	-11.96	-3.32	-0.10	317	325	1216
		0.82x0.82	30.76%	-0.41	-0.09	-0.03	-14.46	-1.20	-0.30	254	262	1009
bp_be CLK=2.0ns		0.66x0.66	53.26%	-0.04	-0.04	-0.01	-2.65	-1.20	-0.04	401	415	1659
		0.74x0.74	42.83%	-0.04	-0.04	-0.01	-2.35	-1.01	-0.03	342	356	1445
		0.85x0.85	32.25%	-0.03	-0.01	-0.01	-1.49	-0.06	-0.03	336	351	1351
swerv_wrapper CLK=1.0ns	12nm	0.52x0.52	46.85%	-0.11	-0.08	0.01	-18.68	-5.28	0.00	2320	2346	9291
		0.58x0.58	38.14%	0.18	0.10	0.29	0.00	0.00	0.00	2126	2152	8498
		0.67x0.67	30.10%	-0.28	**-0.15**	-0.18	-53.91	-49.30	-47.37	2897	2923	9775
coyote CLK=3.2ns		0.56x0.56	47.87%	-0.09	-0.01	0.01	-0.80	-0.01	0.01	3455	3514	13804
		0.59x0.59	42.64%	-0.15	-0.05	-0.02	-2.87	-0.12	-0.02	3986	4045	15872
		0.68x0.68	38.33%	-0.14	-0.09	0.00	-1.07	-0.48	0.00	3698	3757	14715

compare our post-DR outcomes against the flows in Fig. 3.5(a) and (b). Tables 3.5 and 3.6 compare post-DR WS and post-DR TNS for the four flows applied in OpenROAD and in a commercial tool. The ground-truth flow optimizes paths that are critical as per the DR-based parasitics, while the traditional flow optimizes paths that may or may not be critical post-DR. For example, Fig. 3.12 shows a timing path from OpenSTA after DR. At left, we see the post-DR timing path from the traditional flow, and at right, we list the same path from the ML-based flow. The post-GR timing optimization does not have an accurate estimation of the path that becomes critical after DR and hence does not have correct resizing or buffering, while our ML-based flow identifies this as a critical path during post-GR optimization, and resizes the gate and buffers the net to ensure that the path has better timing after DR. As a consequence of better optimization, we improve post-DR WS.

Table 3.5 compares the post-DR WS and TNS from the ML-based flow and from the traditional flow, in OpenROAD. Out of 15 designs, the ML-based flow improves post-DR WS for 13 designs and improves post-DR TNS for 14 designs as highlighted in green, indicating effective timing optimization post-GR using the ML model. Of the remaining cases, "Trad." and "ML" achieve the same results. For the cases in bold green, the post-DR results from the ML-based flow are even better than the results from the ground-truth-based flow. Based on a detailed analysis of the results, we attribute this to two reasons. First, the path slack from the ML-based estimation can be pessimistic due to prediction error; this makes the tool flow optimize delays more aggressively during post-GR optimization and thus helps improve post-DR WNS. Second, the routing of

```
Startpoint: _81642_ (rising edge-triggered flip-flop clocked by CLK)
Endpoint: fe_cmd_o[51] (output port clocked by CLK)
Path Group: CLK
Path Type: max

   Cap     Slew    Delay    Time     Description
--------------------------------------------------------------------------------
                   0.0000   0.0000   clock CLK (rise edge)
                   0.3078   0.3078   clock network delay (propagated)
          0.0238   0.0000   0.3078 ^ _81642_/CK (DFF_X2)
14.1420   0.0195   0.1322   0.4399 ^ _81642_/Q (DFF_X2)
          0.0195   0.0006   0.4406 ^ repeater7842/A (BUF_X16)
59.0805   0.0101   0.0273   0.4678 ^ repeater7842/Z (BUF_X16)
   :        :        :        :        :
          0.0058   0.0003   0.7403 ^ repeater5814/A (BUF_X4)
22.7279   0.0151   0.0286   0.7689 ^ repeater5814/Z (BUF_X4)
          0.0152   0.0019   0.7708 ^ repeater5813/A (BUF_X4)
23.5321   0.0155   0.0325   0.8033 ^ repeater5813/Z (BUF_X4)
          0.0156   0.0020   0.8053 ^ repeater5812/A (BUF_X8)
24.1918   0.0098   0.0266   0.8318 ^ repeater5812/Z (BUF_X8)
   :        :        :        :        :
          0.0142   0.0003   1.1063 v _44515_/B (MUX2_X1)
 1.5798   0.0099   0.0600   1.1663 v _44515_/Z (MUX2_X1)
          0.0099   0.0000   1.1663 v _44516_/C2 (AOI221_X1)
 1.8059   0.0421   0.0547   1.2210 ^ _44516_/ZN (AOI221_X1)
          0.0421   0.0000   1.2210 ^ _44535_/A1 (NAND3_X1)
 3.1779   0.0195   0.0320   1.2530 v _44535_/ZN (NAND3_X1)
   :        :        :        :        :
          0.0161   0.0013   1.4471 ^ _45105_/A2 (NAND3_X4)
12.8435   0.0153   0.0268   1.4739 v _45105_/ZN (NAND3_X4)
          0.0153   0.0005   1.4744 v repeater3042/A (BUF_X4)
28.7790   0.0101   0.0354   1.5098 v repeater3042/Z (BUF_X4)
          0.0106   0.0028   1.5127 v repeater3038/A (BUF_X4)
31.7838   0.0105   0.0329   1.5456 v repeater3038/Z (BUF_X4)
   :        :        :        :        :
          0.0069   0.0005   2.2568 v _46427_/B1 (OAI22_X4)
26.8284   0.0531   0.0607   2.3176 ^ _46427_/ZN (OAI22_X4)
          0.0532   0.0065   2.3241 ^ output1779/A (BUF_X1)
 1.0893   0.0074   0.0291   2.3532 ^ output1779/Z (BUF_X1)
          0.0074   0.0000   2.3533 ^ fe_cmd_o[51] (out)
--------------------------------------------------------------------------------
                   2.2000   data required time
                  -2.3533   data arrival time
--------------------------------------------------------------------------------
                  -0.1533   slack (VIOLATED)
```

```
Startpoint: _81642_ (rising edge-triggered flip-flop clocked by CLK)
Endpoint: fe_cmd_o[51] (output port clocked by CLK)
Path Group: CLK
Path Type: max

   Cap     Slew    Delay    Time     Description
--------------------------------------------------------------------------------
                   0.0000   0.0000   clock CLK (rise edge)
                   0.2804   0.2804   clock network delay (propagated)
          0.0237   0.0000   0.2804 ^ _81642_/CK (DFF_X2)
14.1118   0.0195   0.1321   0.4126 ^ _81642_/Q (DFF_X2)
          0.0195   0.0006   0.4132 ^ repeater7842/A (BUF_X16)
59.4902   0.0102   0.0274   0.4406 ^ repeater7842/Z (BUF_X16)
   :        :        :        :        :
          0.0057   0.0003   0.7070 ^ repeater5814/A (BUF_X4)
26.1237   0.0168   0.0302   0.7372 ^ repeater5814/Z (BUF_X4)
          0.0170   0.0027   0.7398 ^ repeater5813/A (BUF_X8)
23.5811   0.0097   0.0268   0.7666 ^ repeater5813/Z (BUF_X8)
          0.0097   0.0019   0.7685 ^ repeater5812/A (BUF_X8)
23.9374   0.0097   0.0245   0.7930 ^ repeater5812/Z (BUF_X8)
   :        :        :        :        :
          0.0142   0.0003   1.0677 v _44515_/B (MUX2_X1)
 3.0629   0.0114   0.0639   1.1316 v _44515_/Z (MUX2_X1)
          0.0114   0.0001   1.1317 v _44516_/C2 (AOI221_X2)
 1.8426   0.0360   0.0482   1.1799 ^ _44516_/ZN (AOI221_X2)
          0.0360   0.0000   1.1800 ^ _44535_/A1 (NAND3_X1)
 3.0302   0.0179   0.0298   1.2097 v _44535_/ZN (NAND3_X1)
   :        :        :        :        :
          0.0162   0.0013   1.4030 ^ _45105_/A2 (NAND3_X4)
15.3944   0.0167   0.0287   1.4318 v _45105_/ZN (NAND3_X4)
          0.0167   0.0006   1.4324 v repeater3042/A (BUF_X8)
28.3560   0.0071   0.0314   1.4637 v repeater3042/Z (BUF_X8)
          0.0077   0.0032   1.4669 v repeater3038/A (BUF_X4)
32.1545   0.0106   0.0316   1.4985 v repeater3038/Z (BUF_X4)
   :        :        :        :        :
          0.0069   0.0005   2.2049 v _46427_/B1 (OAI22_X4)
26.8384   0.0533   0.0614   2.2662 ^ _46427_/ZN (OAI22_X4)
          0.0534   0.0057   2.2719 ^ output1779/A (BUF_X2)
 1.0229   0.0066   0.0262   2.2982 ^ output1779/Z (BUF_X2)
          0.0066   0.0000   2.2982 ^ fe_cmd_o[51] (out)
--------------------------------------------------------------------------------
                   2.2000   data required time
                  -2.2982   data arrival time
--------------------------------------------------------------------------------
                  -0.0982   slack (VIOLATED)
```

Figure 3.12: A critical path from bp_be 45nm after DR, from the traditional flow (left) and from the ML-based flow (right).

the worst post-DR timing path of the ground-truth-based flow has larger detours than the routing of the same path in the ML-based flow, which leads to more parasitics and worse timing, even though the two flows achieve the same buffering and sizing during post-GR optimization.

Table 3.5 also compares the ML-based flow to the flow proposed in the preliminary version of this work in [40]. We find that the post-DR WS of 10 designs and the post-DR TNS of 13 designs are improved by our introduction of new features.

To determine the impact of ML-based post-GR timing optimizations on routability, we analyze congestion under traditional and ML-based flows. We define a GCell to be congested if its congestion exceeds a specified threshold. Fig. 3.13 shows the number of congested GCells in the traditional and ML-based flows, for different thresholds. The ML-based model does not increase the number of congested GCells (i.e., the traces are superposed), indicating that it does not impact routability.

The last three columns of Table 3.5 compare the runtimes of three different flows. We see that the ML-based flow can leverage prediction of post-DR timing at the cost of a few tens of seconds, without time-intensive DR. The ML-based flow achieves comparable solutions to the ground-truth-based flow, with an average speedup of 31.87% over the ground-truth flow. When compared to the traditional flow, the ML-based flow is only marginally (0.84%) slower than the traditional flow, which is acceptable as the model can achieve a quality that is closer to, and sometimes better than, the ground-truth flow.

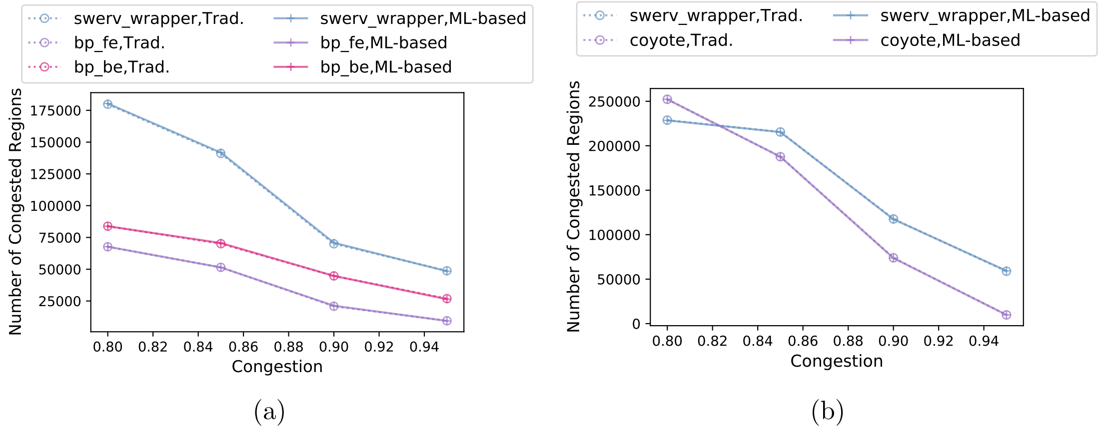


Figure 3.13: Number of post-DR congested regions in the traditional and ML-based flows in (a) 45nm designs and (b) 12nm designs.

Table 3.6 compares the post-DR WS and TNS results from the commercial tool flow. Out of 15 designs, the ML-based flow improves post-DR WS for 10 designs and improves post-DR TNS 13 designs. Four designs have the same post-DR WS in the traditional flow and the ML-based flow. One design (swerv_wrapper in 12nm) improves the WS of 0.18ns from the traditional flow to 0.10ns using our approach, with lower buffering and sizing cost. For post-DR TNS, the ML-based flow shows better results in 13 designs, and the remaining designs do not have path violations.

3.5.3 Generalization to different clock periods

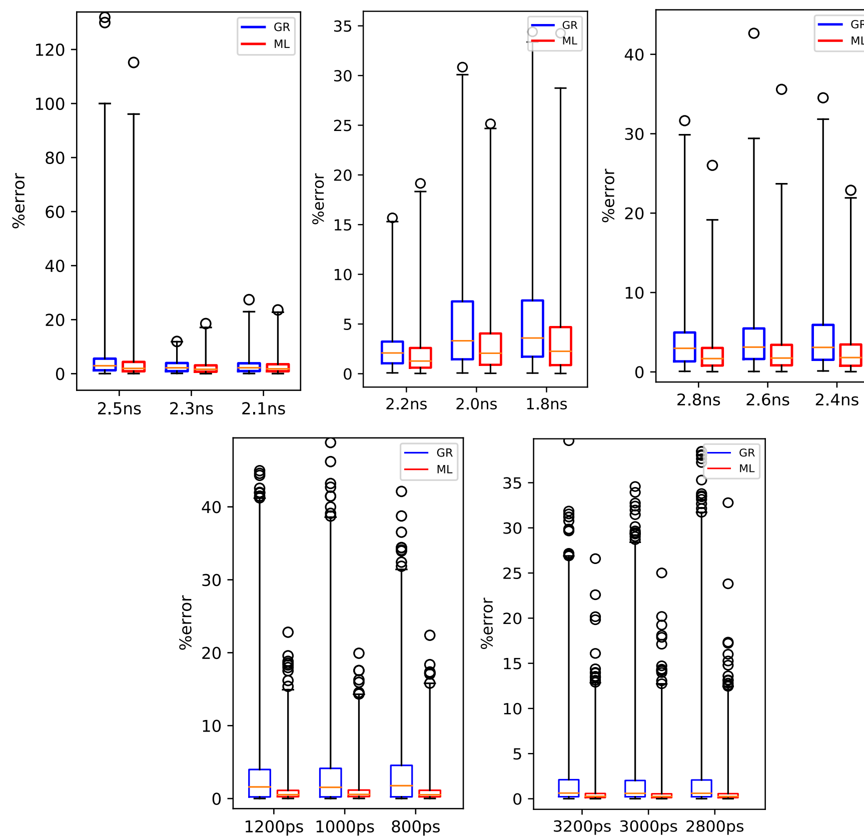


Figure 3.14: Sensitivity of wire delay models to different clock periods for (from left to right) swerv_wrapper, bp_fe, bp_be in 45nm, and swerv_wrapper, coyote in 12nm.

In the experiments above, the designs used for training have been implemented at a single target clock period. We now analyze the ability of our ML models to generalize

Table 3.7: Mean %error of prediction for designs generated using different clock periods.

Design	Tech	Clock period	ML wire delay	GR wire delay	ML wire slew	GR wire slew
swerv_wrapper	45nm	2.50ns	0.84%	4.27%	0.04%	0.20%
		2.30ns	2.40%	4.50%	0.06%	0.06%
		2.10ns	3.35%	5.61%	0.07%	0.09%
bp_fe		2.20ns	0.64%	3.63%	0.04%	0.24%
		2.00ns	2.91%	3.15%	0.23%	0.31%
		1.80ns	2.40%	3.78%	0.16%	0.25%
bp_be		2.80ns	2.16%	3.00%	0.07%	0.12%
		2.60ns	2.33%	3.13%	0.12%	0.13%
		2.40ns	2.51%	3.70%	0.11%	0.13%
swerv_wrapper	12nm	1.20ns	0.87%	4.26%	0.04%	0.70%
		1.00ns	1.21%	4.03%	0.12%	0.56%
		0.80ns	1.22%	4.79%	0.26%	1.31%
coyote		3.20ns	0.60%	3.47%	0.22%	1.97%
		3.00ns	0.75%	3.23%	0.32%	1.99%
		2.80ns	0.81%	3.17%	0.31%	1.68%

with respect to unseen designs generated by using different clock periods from the one used for training. To test the sensitivity of the models to different clock periods, we implement each design with two more clock periods and extract the input features for running ML inferences on two unseen designs. Fig. 3.14 shows the prediction accuracy for all designs generated using three different clock constraints. Here, the model presented in Tables 3.4 and 3.5 is used for the leftmost clock period. The leftmost clock period of each plot is used for training, which is also denoted in blue in Table 3.7, and the other two clock periods are unseen during the training. The mean %error for each case is listed in Table 3.7. In the box-whisker plot, the boxes indicate the 25th percentile, 50th percentile (the median, shown by a yellow line), and 75th percentile of the prediction error of the traditional method in GR and ML-based method. For the ML-based method, the box and the median are generally lower than for the traditional method, indicating that ML models have good ability to generalize with respect to unseen designs with different clock constraints. The table summarizes the mean %error of the prediction for different clock periods, one of which used for training and is highlighted in blue. The smaller ML prediction errors for designs in 12nm also indicate that our models generalize well in 12nm.

Table 3.8: Impact of training-stage noise, for various values of the noise standard deviation, on prediction accuracy.

Design	Tech	Metrics	Wire Delay			Wire Slew		
			Trad.	ML	ML-Noise(0.01, 0.05, 0.1)	Trad.	ML	ML-Noise(0.01, 0.05, 0.1)
swerv_wrapper clk=2.5ns	45nm	Mean %error	4.27%	0.84%	(1.37%, 1.89%, 2.65%)	0.20%	0.04%	(0.09%, 0.20%, 0.33%)
		Max %error	149.41%	115.19%	(124.90%, 133.41%, 137.82%)	40.63%	19.81%	(33.79%, 33.76%, 34.38%)
		Std. dev. %error	7.28%	2.78%	(4.34%, 5.24%, 5.73%)	1.22%	0.34%	(0.65%, 0.76%, 0.75%)
bp_fe clk=2.2ns		Mean %error	3.63%	0.64%	(1.60%,4.39%, 7.06%)	0.24%	0.04%	(0.10%, 0.31%, 0.51%)
		Max %error	15.68%	19.14%	(25.32%, 27.98%, 47.65%)	5.83%	4.42%	(3.70%, 5.36%, 7.39%)
		Std. dev. %error	4.65%	0.91%	(1.47%, 4.17%, 7.31%)	0.98%	0.13%	(0.22%, 0.52%, 0.92%)
bp_be clk=2.8ns		Mean %error	3.00%	2.16%	(2.64%, 2.71%, 4.36%)	0.12%	0.07%	(0.08%, 0.19%, 0.33%)
		Max %error	55.64%	34.58%	(31.64%, 39.62%, 47.77%)	10.45%	9.15%	(14.62%, 18.69%, 28.10%)
		Std. dev. %error	3.88%	3.42%	(3.78%, 3.08%, 3.23%)	0.38%	0.25%	(0.44%, 0.62%, 0.91%)
swerv_wrapper clk=1200ps	12nm	Mean %error	4.26%	0.87%	(1.03%, 2.73%, 4.89%)	0.70%	0.04%	(0.04%, 0.09%, 0.16%)
		Max %error	44.95%	22.80%	(25.90%, 32.35%, 67.92%)	71.75%	29.81%	(29.28%, 29.65%, 33.14%)
		Std. dev. %error	5.76%	0.93%	(0.99%, 2.69%, 5.25%)	3.85%	0.17%	(0.18%, 0.22%, 0.32%)
coyote clk=3200ps		Mean %error	3.47%	0.60%	(0.68%, 2.34%, 4.16%)	1.97%	0.22%	(0.23%, 0.80%, 1.43%)
		Max %error	39.65%	26.58%	(27.65%, 30.56%, 63.40%)	51.34%	40.13%	(41.80%, 43.21%, 43.42%)
		Std. dev. %error	4.73%	0.88%	(0.91%, 3.10%, 5.87%)	4.51%	0.62%	(0.64%, 1.10%, 1.78%)

Table 3.9: Impact of noise with various standard deviation on post-DR results.

Design	Tech	Post-DR WS (ns)			Post-DR TNS (ns)		
		Trad.	ML-Noise(0, 0.01, 0.05, 0.1)	Ground-truth	Trad.	ML-Noise(0, 0.01, 0.05, 0.1)	Ground-truth
swerv_wrapper clk=2.5ns	45nm	-0.14	(-0.14, -0.14, -0.14, -0.15)	-0.13	-23.45	(-21.96, -22.58, -23.52, -26.89)	-22.58
bp_fe clk=2.2ns		-0.15	(-0.10, -0.14, -0.16, -0.21)	-0.10	-1.60	(-0.95, -1.40, -1.88, -2.30)	-0.72
bp_be clk=2.8ns		-0.18	(-0.17, -0.17, -0.18, -0.18)	-0.15	-9.63	(-7.82, -7.65, -8.14, -8.77)	-6.60
swerv_wrapper clk=1200ps	12nm	-0.48	(-0.23, -0.28, -0.46, -0.58)	-0.35	-207.68	(-202.38, -207.43,-209.80, -230.48)	-200.55
coyote clk=3200ps		-0.02	(0.49, 0.47, 0.40, 0.34)	0.86	-0.14	(0, 0, 0, 0)	0.00

3.5.4 Noise impact on model performance

As there is inherent noise from EDA tools, the data with noise collected from EDA tools and used in our ML-based flow can affect the DR solution quality. To study the impact of noise on the model accuracy and DR outcomes from the ML-based flow, we add Gaussian noise $\mathcal{N}(0, \sigma)$ to labels and train the models using the datasets with noise. A set of σ values (0, 0.01, 0.05, 0.1) are selected to evaluate the robustness of the models to the noise with different standard deviations. We generate noise matrices that have the same dimension with our training dataset using three different random seeds for each σ value, then add each noise matrix to the training dataset separately. We take the average value of the three predictions from the models trained by the three different training datasets as our prediction result. Then, we compare the results from a modified ML-based flow (where the ML models are replaced by the models trained with noisy data) to the results from the original ML-based flow.

In Table 3.8, the prediction errors of models trained with noisy data are compared to the traditional estimation and the prediction from models trained with a dataset without added noise. According to the results for $\sigma = 0.01$, ML models can handle the noise with $\sigma = 0.01$ value in most cases; the exceptions are maximum %error for wire delay of bp_fe in 45nm, and maximum and standard deviation of %error for wire slew of bp_be in 45nm. These are denoted in red, indicating no improvement compared to the estimation from traditional methods. According to the results for $\sigma = 0.05$, the ML models perform better on four out of five designs for both wire delay and wire delay. However, when $\sigma = 0.1$, which is approximately equal to the maximum wire delay and $2\times$ maximum slew in our datasets, the models trained by noisy data cannot make accurate predictions for most designs.

The impact of noise on post-DR solution quality is summarized in Table 3.9. We compare the post-DR results from the ML-based flow using the models trained with noisy data, to the traditional flow and ground-truth-based flows. The ML-based flow generates better DR solutions than the traditional flow for all testcases when the noise has $\sigma = 0.01$. As we increase the σ value, the DR results of some testcases denoted in red are no longer better than the results from the traditional flow, due to inaccurate timing prediction being used in post-GR timing optimization. For $\sigma = 0.05$ and $\sigma = 0.1$, only two designs out of five designs still see better solutions from the ML-based flow.

3.6 Conclusion

We study an ML-based method to predict post-DR timing at the post-route stage. Our models demonstrate better accuracy of timing and parasitic estimation at the post-GR stage compared to traditional methods, using both OpenROAD and a commercial tool flow. Our approach shows improvements in the DR solution quality; it is shown to be computationally efficient, and does not increase congestion compared to the prior methods. Moreover, our experimental results show that our models generalize well to designs generated using unseen clock periods, and that our flow generates better DR solutions even when models are trained using datasets with noise.

Chapter 4

IR-Aware ECO Timing Optimization Using Reinforcement Learning

4.1 Introduction

Tools and flows for power integrity [43] aim to restrict IR drops below a specified limit, and timing optimization uses gate delay models at this worst-case voltage corner. However, due to limited wiring resources, in late stages of design, the power grid may not meet the IR limit exactly, resulting in increased gate delays and timing failures. By then, the layout is near-final and large perturbations will impede timing closure; only incremental engineering change order (ECO) optimizations, with minimal placement perturbation, are allowable to resolve timing. We address the ECO timing optimization problem to fix timing violations through gate sizing.

Gate sizing for early design stages has been well studied, but there is little work on ECO for IR-induced timing failures. Gate sizing selects a size for every netlist instance from a set of choices in the standard cell library, each with different delay/area/power, and is NP-hard [44]. Early approaches used simple convex models in a space of continuous gate sizes, using sensitivity methods [45], convex programming [46], and Lagrangian relaxation (LR) [47]. The modern version uses more complex nonconvex delay models and discrete gate sizes [48]. ECO-based optimizations in [49, 50] do not address the interaction between IR drop and gate sizing. In [51], ECOs for IR drop are resolved by

moving cells; results show large placement perturbations. Our ECO solution performs low-perturbation sizing to meet timing and is based on reinforcement learning (RL).

RL has been applied to several chip design problems [52, 53]. Prior methods have tackled the related gate sizing problem at early stages of design, without supply voltage awareness [54, 55]. In [54], RL is applied in a black-box framework, and its solution is not suitable for ECO optimization; in [55], imitation learning is used rather than RL, merely accelerating LR methods to explore delay-power Pareto tradeoffs rather than fully harnessing RL.

Prior RL methods for gate sizing cannot be trivially extended to the ECO problem, but it is useful to examine their limitations. *First*, they incur **forbidding runtimes** due to the large action and search spaces for optimization. *Second*, they focus on a **single objective** (e.g., minimizing TNS in [54]), rather than the full multi-objective constrained optimization problem. Using penalty- and weight-based techniques that combine power, area, and timing into a single loss function requires significant parameter tuning per design [56] and is not viable. *Third*, they use RL as a **"black box" optimizer** rather than weaving in prior advances in gate sizing to build a solution that combines the best of traditional and machine learning (ML)-based solvers.

We overcome the first issue through the very nature of our ECO problem formulation: the requirement for minimal perturbation to the existing solution naturally results in action and search spaces of manageable size. To address the second and third issues, we eschew the black-box approach, and instead, leverage domain- and problem-specific knowledge, *using LR to drive RL* by coupling novel RL techniques with essential ideas from prior LR-based approaches. The use of Lagrange multipliers (LMs) from LR also provides a *natural way to solve constrained optimization problems*. The use of problem-specific insights also addresses the first issue by improving the efficiency of RL, which is well known to be sample-inefficient [57].

We propose **RL-LR-Sizer**, coupling deep RL with traditional LR-based techniques for ECO gate sizing. We iteratively answer one of the following questions in each RL iteration: (i) *Order*: "Given a circuit netlist, which gate to operate on (upsize or downsize)?", and (ii) *Choice*: "Given a gate and its local neighborhood, what size to select?" RL-LR-Sizer uses a relational graph convolutional neural network (R-GCN) as an agent to solve the order and choice problems. The agent is trained using deep Q learning [58], interacting with the environment to maximize a reward. We designate a reward function that converts the constrained optimization problem into an unconstrained problem using LMs. The contributions of this work are:

(1) This is the first work to address ECO sizing for library-based NLDM delay models under an RL-based framework.
(2) We couple RL with LR-based techniques to train an R-GCN agent to determine both *order* and *choice*. This naturally enables *multi-objective optimization*, presenting the first RL formulation for constrained gate sizing. We also leverage problem-specific knowledge in gate sizing to help RL find better solutions.
(3) Our novel clock update method (Sec. 4.4) during training enhances model quality over a range of timing specifications.
(4) We train the RL-LR sizer and use it for inference for ECO changes at multiple timing specifications across multiple designs. We show (a) a full training flow per design; (b) zero-shot inference flow using a trained model on an unseen timing specification or unseen design; and (c) fine tuning flow on a trained model.

4.2 Problem formulation

The ECO problem is encountered late in the design cycle, after place-and-route and power grid design. Larger-than-expected IR drops result in increased gate delays, and the circuit fails timing. The ECO step resizes devices to bring the circuit back to timing specifications. This involves both device upsizing (to improve drive strength) and downsizing (to reduce the load offered to the previous stage). In principle, upsizing/-downsizing could change the current load and alter the voltage drop, but empirically, the change in current load is small, resulting in supply voltage changes (0.001% of a 1.1V supply).

Formally, the objective of IR-aware ECO timing is to minimize the total power of the design while satisfying performance and electrical constraints after considering post-PD IR drop. The constrained optimization problem is formulated as:

$$\begin{aligned} & \textstyle\sum_{i\in\mathcal{I}} Power_{c_i} \\ \text{subject to} \quad & -\text{slack}_i(V_{dd,i}, GND_i) \leq 0 \;\; \forall i \in \mathcal{I} \\ & c_i \in \mathcal{C}_i \;\; \forall i \in \text{ set of instances } \mathcal{I} \end{aligned} \tag{4.1}$$

where $\mathcal{C}_i$ is the set of choices for instance i in the library; $c_i \in \mathcal{C}_i$ is the library cell for instance i; $Power_{c_i}$ is the power consumption of instance i, computed using a vectorless approach and including internal, switching, and leakage power; and slack_i is the slack at instance i as a function of post-PD rail voltages, $V_{dd,i}$ and GND_i.

Problem (4.1) may be solved using penalty functions [59], minimizing a weighted sum of the objective and the constraint violations.

4.3 RL-LR ECO timing framework

The RL-LR ECO timing framework solves (4.1) by:
(a) *Representing* the netlist as a graph with node-level features;
(b) *Mapping* the sizing problem to an RL-solvable control problem;
(c) *Developing* a deep Q-network (DQN) framework, coupled with an LM update strategy, for training the R-GCN model (RL agent).

4.3.1 The circuit netlist as an annotated graph

The circuit netlist is represented as a directed relational graph $G = (\mathcal{V}, \mathcal{E}, \mathcal{R})$ where $\mathcal{V}$ is a set of all nodes representing the instance in the design, $\mathcal{E}$ is a set of all edges representing the nets in the design, and $\mathcal{R}$ is a set of all possible relation types that represent the input or output relation of each edge to the node in the graph. We convert the hyperedges in the circuit to a star representation [60] where each wire from instance i to instance j is represented by a edge $(v_i, r, v_j) \in \mathcal{E}$ with relation r. The graph representation of the netlist allows us to solve the gate sizing problem through the use of deep RL algorithms.

The nodes in the graph are annotated by a list of features (Table 4.1), used by the R-GCN agent that are capable of understanding the relations between nodes to make decisions. In our work, R-GCNs understand edge directions as the gate delay is dependent on identifying the driver and loads. R-GCNs accumulate annotated features from the set of neighbors, $\mathcal{N}_i^r$, of instance i under relation $r \in \mathcal{R}$. From outgoing edges and incoming edges, R-GCN aggregates different features from neighbors. The output load capacitance is only aggregated from outgoing edges, and the input slew is only aggregated from incoming edges. All other features are aggregated from both edges. The slack, slew, load and IR voltage drop features help the R-GCN agent make decisions that meet timing constraints. The instance type feature enables the agent's decisions to minimize power and meet timing.

Table 4.1: List of the annotated features on instance i in G.

Feature	Definition
slack_i	Slack at the output pin.
in_slew_i	Maximum slew at the input pins.
out_slew_i	Slew at the output pin.
instance_type	Size and type of cell.
load_i	Capacitive load at the output pin.
ir_voltage_i	Power supply voltage at th cell with IR drop considerd.

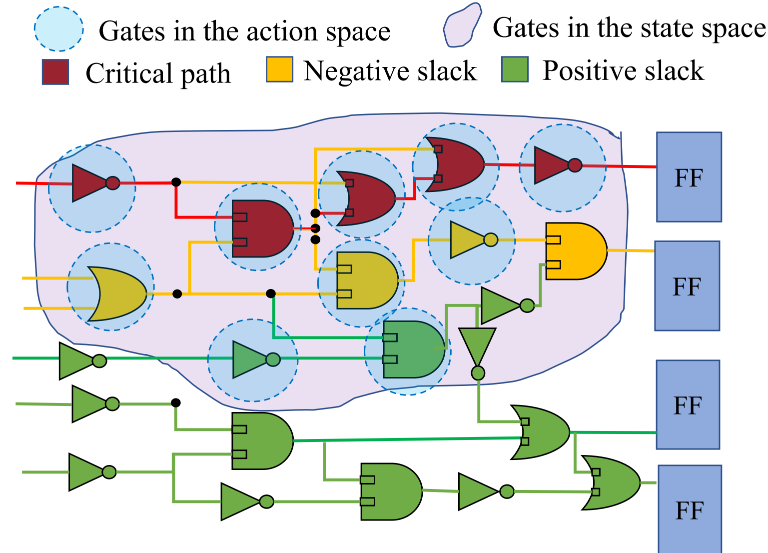


Figure 4.1: State and action spaces from the circuit netlist.

4.3.2 Mapping IR-aware gate sizing to RL

We map the sizing problem (4.1) to a Markov Decision Process (MDP), making sequential RL-based decisions. We define the following:

State: The state s_t at step t is a subgraph of the annotated graph G that contains all instances v_i that have $\text{slack}_i < 0$, or lie within a two-hop neighborhood[1] of any instance with $\text{slack}_i < 0$. Unlike [54], which uses a local embedding of a single independent instance and its three-hop neighborhood as the state, we use ***all*** netlist instances with negative slack and their neighborhood as the state, letting the RL agent decide which instances in the state to operate on.

Fig. 4.1 illustrates the state s_t at time step t (on this toy example, s_t is a large fraction

[1]A two-hop neighborhood is sufficient as the timing impact the choice of a gate size has on the netlist diminishes with the increase in the hops.

of the circuit, but for a large circuit [des_perf. 21k gates], s_t contains *only 0.06% of the gates*). The red instances correspond to gates on the critical path (most negative slack) after considering IR drop impact, the yellow instances correspond to gates that also have negative slack (near-critical paths) and the gates in the purple region are those that belong to the two-hop neighborhood of any instance with $\text{slack}_i < 0$ (red or yellow instances). This state representation provides the RL agent with a global view of the graph, preventing the optimizer from being stuck within local minima. In contrast, [48,54,55] operate with a single instance (the instance being sized in the current iteration), and its immediate neighborhood as the state. Mimicking these methods would provide an RL agent with little local information, leading to local minima.

Action: We define an action as both the order (which gate is selected) and choice (whether it is upsized or downsized). Instead of using all nodes in the state as the action space, we prune the space for faster convergence of RL training by creating an action mask:

$$\phi(x_a) = \begin{cases} x_a & \text{if } a \text{ is a valid action} \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

The mask constrains the agent to select gates that are on, or within a two-hop neighborhood of any gate on the critical path. The size of the action space, which is based on the *critical path* only, is a small fraction of the state space for large circuits. The mask also prevents invalid actions, e.g., upsizing a gate at its largest available size. In Fig. 4.1, all instances highlighted in dotted circles are within the action space. The action space size is twice the number of candidate gates, as each gate can be either upsized or downsized.

Based on the state and action spaces, we set up the gate sizing RL formulation. However, the action space is large, creating challenges for the already sample-inefficient RL algorithm [57]. *We limit the size of the action space by coupling RL training with problem-specific knowledge from LR-based gate sizing, as described Section 4.4.2.*

Reward: We use **domain-specific knowledge** to create the reward function: instead of using an arbitarily-weighted penalty function to translate (4.1) to an unconstrained objective, we leverage prior work on Lagrangian relaxation for gate sizing [48] to create an unconstrained reward function. Using nonnegative Lagrange multipliers (LMs), λ_i, for each slack constraint, we minimize:

$$\boldsymbol{L_{\lambda}}(c, \text{slack}) : \textstyle\sum_{i \in \mathcal{I}} Power_{c_i} + \sum_{i \in \mathcal{I}} L_{slack,i} \tag{4.3}$$

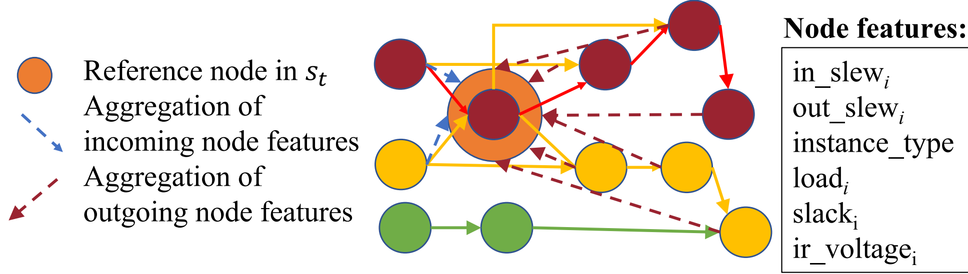


Figure 4.2: R-GCN aggregation from a 3-hop neighborhood.

$$\text{where } L_{\text{slack},i} = \begin{cases} \frac{\lambda_i(-\text{slack}_i)}{\beta \times \text{TNS} + \epsilon_0} & \text{if TNS} \leq \alpha \times \text{clk} \\ \lambda_i(-\text{slack}_i) & \text{otherwise} \end{cases} \tag{4.4}$$

and $\boldsymbol{\lambda}$ is the vector of λ_is. Here, α and β are tunable constants and $\beta < 0$, ϵ_0 is a small value to prevent divide-by-zero; clk is the clock period; TNS is the total negative slack; and c [slack] is a set variable for c_i [slack$_i$]. We formulate $L_{\text{slack},i}$ to provide a higher importance to the slack component of the Langrangian for slacks near zero.

We will solve the unconstrained Lagrangian relaxation subproblem (LRS) iteratively, updating the LMs $\boldsymbol{\lambda}$, as described in Section 4.3. We embed the LRS objective into the EL reward function:

$$R_t = L^t_{\lambda}(c, \text{slack}) - L^{t+1}_{\lambda}(c, \text{slack}) \tag{4.5}$$

where $L^t_{\lambda}(c, \text{slack})$ and $L^{t+1}_{\lambda}(c, \text{slack})$ is the value of the LRS-based objective function at steps t and $t+1$, respectively, due to action a_t. Since a_t changes the size of only one gate, with every action, we perform an incremental timing update to evaluate $L^{t+1}_{\lambda}(c, \text{slack})$.
RL agent (R-GCN): GCN-based agents that work on a graph create effective representations of the circuit that encode features into a meaningful embedding through a message passing scheme.

The aggregation for the ECO sizing problem is shown in Fig. 4.2, for the reference node highlighted in orange, corresponding to the state in Fig. 4.1. The relation r in eq.(2.3) is the edge direction in our R-GCN. *The use of this R-GCN, rather than a GCN, is critical in capturing the direction of timing flow from input to output of a gate.* For $\sigma(.)$ activation function, we use ReLU. This propagation model is applied to subgraph s_t to generate $h_i(l+1)$ in layer $l+1$.

Fig. 4.3 shows the structure of the R-GCN agent, with three R-GCN layers, i.e., each node aggregates features from a three-hop neighborhood. The first two layers, each of

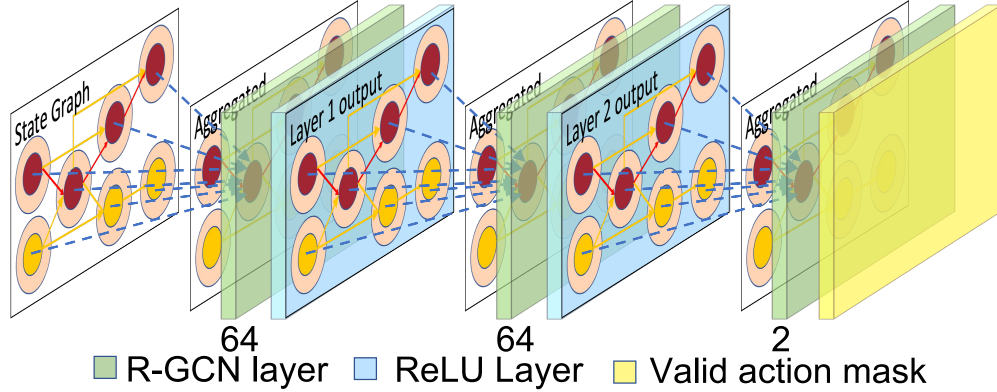


Figure 4.3: Structure of the three-layered R-GCN agent.

dimension 64, are followed by ReLU activation functions. The last layer is fed to the action mask to generate a valid action that maximizes the reward.

Environment (Env): The environment includes components that interact with the agent: in our case, this is the inbuilt timing engine that estimates the reward and updates the graph based on an action.

4.3.3 Application of the R-GCN agent

We apply our R-GCN agent on a post-P&R netlist. Fig. 4.4 shows the gate sizing flow and how the R-GCN agent interacts with the environment. Based on the netlist graph $G = (\mathcal{V}, \mathcal{E}, \mathcal{R})$, we extract the node-level features to create s_t and the action mask ϕ. The R-GCN agent acts on s_t to perform action a_t. As a consequence of a_t, the environment performs incremental timing analysis, updates the circuit netlist, and computes R_t. A new state is created using the updated netlist and the new features, and this is repeated until TNS, WNS, and power converge.

4.4 RL model training

4.4.1 Core training strategy

We use a deep-Q network (DQN) training algorithm [58] coupled with an LM update strategy [48] to train the R-GCN. In the DQN framework, the R-GCN, known as the Q network ($Q(s, a; W)$), is an approximator for the best action-value function, $Q^*(s, a)$,

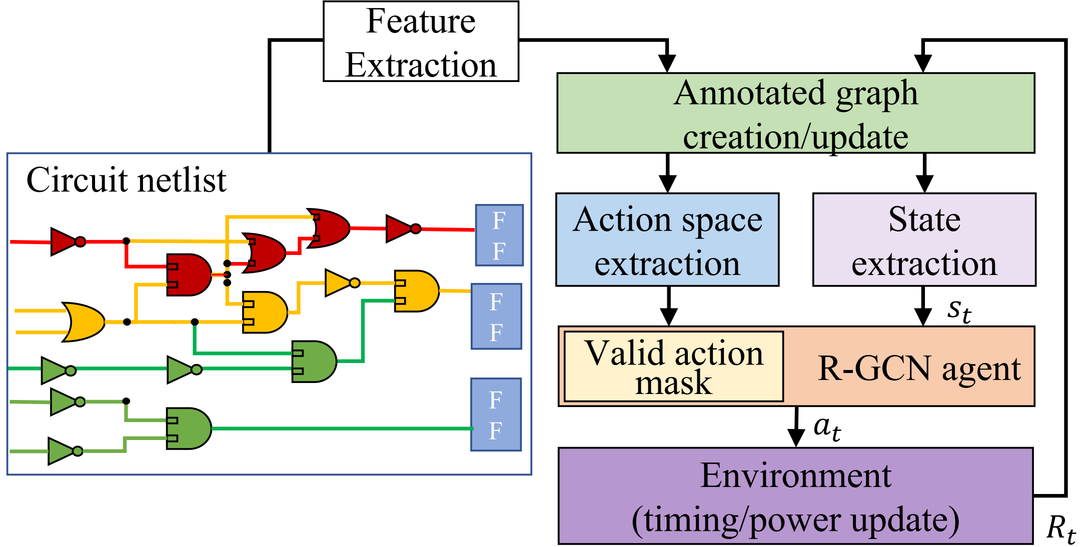


Figure 4.4: Overview of the RL-based ECO gate sizing flow.

to select actions that maximize expected cumulative reward, defined as:

$$Q^*(s,a) = \mathbb{E}\left[R + \gamma \max_{a'} Q^*(s', a'|s, a)\right] \tag{4.6}$$

This expression obeys the Bellman equation, which is based on the idea that if the optimal value $Q^*(s', a')$ in the next time step is known for all possible actions a' from state s', the optimal strategy selects action a' to maximize $\mathbb{E}[R + \gamma Q^*(s', a')]$, where $\mathbb{E}(.)$ is the expected value; future rewards are discounted by γ per time step t.

DQN training: We leverage the DQN training framework described in [58] which begins by initializing a **memory replay buffer** of capacity N, and random initial R-GCN weights. The training iterates over M **episodes**, with each episode containing T timesteps. During each timestep, an action is taken based on an ϵ-greedy policy strategy where it selects an action from the R-GCN with probability $(1 - \epsilon)$ and selects a random action with probability ϵ. Initially, the training begins with a high value for ϵ, encouraging the agent to explore the environment, and decays every episode to a smaller value exploiting the knowledge the agent has gained. We apply the action mask and select an action from the available valid candidates.

Based on the chosen valid action a_t, the environment updates the power and timing incrementally and computes R_t. The transition which includes s_t, s_{t+1}, a_t, and R_t is

stored in a replay buffer at each step. The buffer is sampled at every step to extract a batch of random samples that train the R-GCN policy network. To make training more stable, the DQN training uses a target network, which keeps a copy of R-GCN weights to estimate the $Q^*(s', a')$ value in the Bellman equation. The target network is updated every C episodes by copying the weights from the policy network. The R-GCN policy agent is trained by minimizing the loss function:

$$Loss(W) = \mathbb{E}_{s,a,R}\left[(\mathbb{E}_{s'}[y|s,a] - Q(s,a;W))^2\right] \tag{4.7}$$

where $y = \mathbb{E}_{s'}[R + \gamma \max_{a'} \hat{Q}(s', a', \hat{W})|s, a]$ uses the target network $\hat{Q}(s', a', \hat{W})$ to estimate the discounted future rewards. The target network weights $\hat{W}$ are fixed when optimizing the loss function.

Env reset: We use a modified environment reset strategy in comparison to [58]. At the beginning of each episode, instead of resetting the graph to its original state at the beginning of the first episode, we reset the graph to the state which has the least $\mathbf{L}_\lambda(c, slack)$ in the previous episode. This allows for faster convergence of the objective function during training iterations due to guided explorations.

4.4.2 Problem-specific training enhancement

We incorporate gate-sizing-specific optimizations into the general training framework to enhance the quality of the trained model.

Lagrangian multiplier (LM) update. The use of LMs to determine the relative weights of the components of the cost function is a crucial problem-specific insight used in this work. A direct application of RL would use fixed user-defined weights for λ_i, but instead, we use LR-based update strategies to drive RL exploration. Specifically, we embed the following LM update strategy during DQN training, where we update the values of λ_i every K steps within an episode: $\lambda_i = \lambda_i \times \left(1 - \frac{slack_i}{clock\,period}\right)$. The strategy penalizes large violations more severely than smaller violations. This update strategy helps with guided explorations within the action space. When slack$_i$ is positive or equal to zero, we set the corresponding λ_i to zero.

ECO-based action space reduction. The ECO problem naturally reduces the sizes of the state and action spaces: given a generally good power grid, it is likely that only some combinational blocks of the sequential circuit in IR-affected regions will require ECOs. Only gates in these regions will be part of the RL formulation.

Clock constraint update. The RL framework provides a larger positive reward when

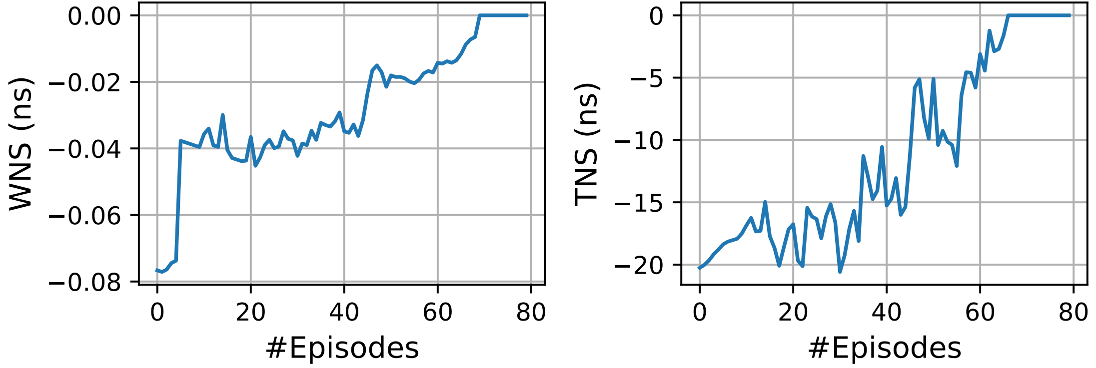


Figure 4.5: Results of training, showing the convergence of the WNS and TNS.

the goal is met. A naïve RL implementation would provide this reward when the timing specification is met – but for tight constraints, the RL may find it difficult to reach the specification. To counter this, recognizing that the delay is progressively reduced using sizing operations, we provide a set of intermediate timing levels as goals that achieve this larger reward, progressively tightening the goal until the delay reaches the specification.

In the first M_{decay} of M total episodes, the clock constraint decays linearly: at the start of each episode, the clock constraint is reduced by $\frac{(\text{initial_delay}-\text{target_delay})}{\text{M}_{decay}}$, where initial_delay is the delay in post-placement and post-PDN synthesis with IR-drop impact considered and target_delay is the required delay. After M_{decay} episodes, the clock constraint reduces to target_delay, and is kept unchanged in the remaining training episodes. We set $\text{M}_{decay} = 30$.

To facilitate transferability across multiple designs and start points, we encapsulate away design-specific information and clock information into the state variable. We work with the lambda slack sum, i.e. $\sum_{i\in\mathcal{I}} L_{slack,i}$ in eq. (4.3), as the loss function, where the slacks are normalized by clock value set in each episode, instead of absolute delay values, which may vary from circuit to circuit.

Fig. 4.5 shows the effectiveness of these approaches in training for circuit wb_conmax. After M episodes, we see that TNS and WNS have converged. Once trained, the R-GCN agent (target network) is applied to the circuit netlist (see Section 4.3.3) to sequentially select actions that solve the constrained optimization problem.

4.5 Evaluation of RL-LR ECO

4.5.1 Experimental setup

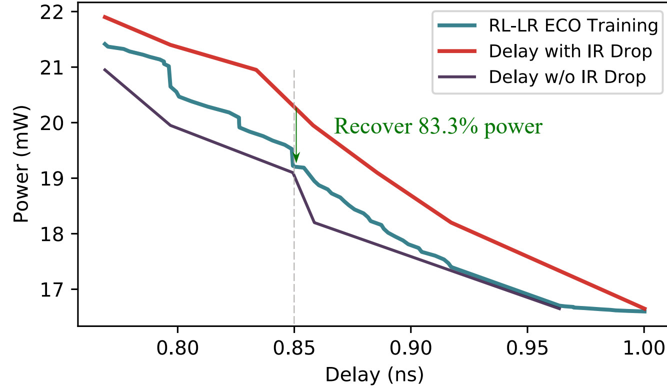


Figure 4.6: Power-delay tradeoff curves for wb_conmax in a 45nm technology, showing a shift to the right due to IR drop, and the results of our RL-LR ECO approach.

We use Design Compiler to size gate-level netlists under various timing constraints at the supply voltage corner. We use OpenROAD [41] to place the design, synthesize the power delivery network, and analyze the static IR drop. An example delay-power curve for wb_conmax is shown as "Delay w/o IR-drop" in Fig. 4.6. The PDN is built to so that the maximum IR drop is low (∼5mV) or moderate (∼10mV) scenarios where delay degradations can be recovered through ECOs. To quantify delay shifts due to IR drop, we annotate NLDM entries of each library cell with its delay sensitivity to voltage drop, and compute circuit delays using the OpenSTA timer. Under IR drop, the delay-power curve shifts to the right, from the "Delay w/o IR drop" to the "Delay with IR drop" curve in Fig. 4.6.

The RL-LR ECO timing (training and application) flow is built using Python libraries, including PyTorch and DGL [61]. We use SWIG-based Python enablements for incremental timing analysis, logic gate swap, and pin and cell property from OpenROAD APIs, which integrates easily into Env for reward computation and state transitions. OpenROAD eliminates challenges of slow TCL interfaces between commercial tools and Python environments in [54]. We train the R-GCN agent using the hyperparameters $M = 50$; $N = 4000$; $T = 75$; $\gamma = 0.99$; $\alpha = 10$; $\beta = -0.1$; $C = 25$; $K = 30$.

Our testcases are 7 OpenCores benchmarks using an open 45nm technology, with

Table 4.2: IR-induced Timing Failures

Design	des_area	wb_dma	pid_controller	aes_cipher_top	des_perf	pci_bridge32	wb_conmax
Target period (ns)	0. 56	0.47	0.75	0.88	0.59	0.76	0.80
#Instances on failed paths (5mV max. IR)	25	36	78	528	492	220	318
#Instances on failed paths (10mV max. IR)	129	226	182	1482	2110	1142	1530

2k–26k gates. Table 4.2 shows the number of instances on paths that fail timing. We implement multiple flows:

(1) **RL-LR ECO Training** (Table 4.3) trains an R-GCN model from scratch, starting from the initial delay, until the target delay is met.

(2) **RL-LR ECO Inference** uses a model that is trained across a wide range of timing constraints to obtain the full delay-power Pareto curve. This training step is applicable for larger delay reductions than ECO optimizations, but requires more computation than that for ECO training. Given this trained model, we report:

(a) **Inference across Timing Constraints (Inf-TC)** (Table 4.4) shows transferability for inference across multiple timing specifications, using a model trained on the same design.

(b) **Inference across Designs (Inf-D)** (Table 4.5) demonstrates transferability across designs, running inference for multiple designs using the model trained with four selected designs.

(c) **Fine Tuning** (Table 4.5) applies inexpensive design-specific fine tuning to the "Inference across Designs" model, incrementally updating weights of the pretrained R-GCN model.

(3) **LR Baseline** (Tables 4.3–4.5) implements a conventional LR flow [48] against which the RL methods are compared.

All runtimes are reported on an Intel Xeon Silver 4214 CPU @2.2GHz and NVIDIA A100 PCIe 40GB GPU.

4.5.2 Optimization and delay-power tradeoffs

As stated earlier, Fig. 4.6 shows the power-delay tradeoff for the ideal IR drop ("Delay w/o IR drop") and the actual IR drop ("Delay with IR drop") for the circuit wb_conmax.

Table 4.3: ECO results for three flows: LR baseline, inference across timing constraints (Inf-TC), and RL-LR ECO training

Designs	Target Delay (ns)	Initial Delay (ns)	% Power Overhead Savings			Runtime(s)			# of Upsize/Downsize		
			LR	Inf-TC	RL-LR ECO Training	LR	Inf-TC	RL-LR ECO Training	LR	Inf-TC	RL-LR ECO Training
des_area	0.56	0.590	35.2%	37.0%	48.8%	82	23	2651	29/0	36/6	27/4
wb_dma	0.47	0.486	16.7%	18.9%	19.7%	47	42	2284	12/0	19/0	18/3
pid_controller	0.75	0.781	32.6%	33.3%	33.7%	58	21	2881	29/2	29/2	20/9
aes_cipher_top	0.89	0.941	8.6%	13.8%	27.2%	654	153	2305	386/3	301/3	291/1
des_perf	0.60	0.632	8.3%	15.6%	12.8%	576	398	2322	491/1	393/1	380/5
pci_bridge32	0.75	0.781	16.0%	16.0%	16.2%	92	54	2336	24/1	22/1	23/1
wb_conmax	0.77	0.834	11.8%	39.5%	45.3%	290	242	3193	86/0	78/0	73/0
Average			18.4%	24.9%	29.1%		−44.5%			+3.3%	−7.8%
Maximum			35.2%	39.5%	48.8%		−76.6%			−22.0%	−31.0%

The third curve shows the result of our RL-LR ECO algorithm, which is seen to recover the delay degradation due to IR drop. To quantify the effectiveness of ECO, we define a recovery metric: for a specific delay D, we define the power overhead savings of the RL-LR ECO approach as $\frac{\text{Power_IR}(D)-\text{Power_RL-LR}(D)}{\text{Power_IR}(D)-\text{Power_w/o_IR}(D)}$, where "IR," "RL-LR," and "w/o_IR" refer to the three curves in the figure. For example, at $D = 0.85$ns, 83.3% of the IR-induced power overhead is recovered.

Table 4.3 compares the traditional LR baseline, RL-LR ECO Training, and Inf-TC for the seven benchmark designs under moderate IR drops, for a specified target delay. The inference model is trained just once and has a training time of 40–70 minutes. The training cost can be amortized over multiple ECO explorations during late-stage design, e.g., perturbations to routing, PDN design, or placement; each translates to altered slack, and is covered by our formulation.

The table shows the initial delay of the circuit, i.e., the delay under the ideal IR drop; the total power and runtimes for the three approaches; and the number of cell changes using the LR and RL-LR ECO training methods. At each target delay, the RL-LR ECO training saves as much or more power than the LR baseline flow, with 29.1% average and 48.8% maximum savings. The inference approach performs slightly worse than RL-LR ECO training in six testcases, but better in one testcase because the solution (a sequence of sizing) from RL-LR ECO training flow mixes a few random actions for exploration purposes during training. All inferences provide solutions of similar quality to LR baseline, with some improvements.

Table 4.3 also reports the number of upsized/downsized cells: downsized cells stay in place and do not require further layout changes, but upsized cells require placement

Table 4.4: Results on multiple clock constraints for wb_conmax

Designs	Target Delay (ns)	Initial Delay (ns)	% Power Overhead Savings		
			LR	Inf-TC	RL-LR ECO Training
wb_conmax	0.77	0.834	11.76%	39.5%	45.3%
	0.79	0.858	8.0%	10.3%	12.8%
	0.84	0.886	71.7%	94.3%	88.8%
	0.86	0.917	40.3%	60.5%	50.2%
	0.96	1.0002	71.5%	95.2%	90.1%
Average			40.7%	60.0%	57.5%
Maximum			71.7%	95.2%	90.1%

perturbations to remove overlaps. Both Inference and RL-LR ECO training upsize fewer cells than LR for large designs, thus easing timing closure.

As is typical of RL-based approaches [54], RL-LR ECO training has very high runtime as it includes R-GCN training and a reward computation (STA update) during each training step. The Inf-TC model, with offline training, shows an average of 44.5% runtime reduction over LR. After one-time training, the trained model can be applied to any operating period on the delay-power curve.

To demonstrate that the model is transferable across multiple start points within a design, Table 4.4 presents results for five start points for wb_conmax. The result for the Inference flow is similar to the RL-LR training from scratch, and better than the LR baseline.

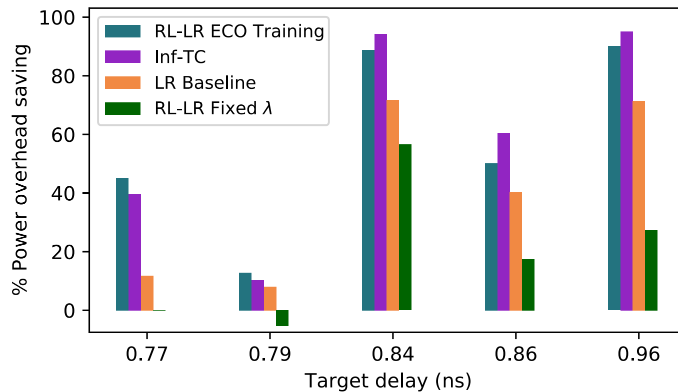


Figure 4.7: Power overhead savings for wb_conmax.

4.5.3 Importance of LR-based weight adaptation

We highlight crucial aspects of the RL-LR ECO optimizer: (i) guided explorations through LR, and (ii) solving both the gate order and choice (gate size selection) problems, as defined in Section 4.1.

Fig. 4.7 shows power overhead savings for wb_conmax, at different target delays, for several flows: our trained RL-LR ECO approach, the classical LR baseline, Inf-TC, and RL-LR with fixed weights, λ. Across all target delays, our RL-LR ECO optimizer and Inf-TC methods show better savings than the classical LR solution.

To understand the importance of using LMs to determine the relative weights of the objective function components, dynamically changed during the RL optimization, we consider the use of fixed λ_i penalties for the slack term in (4.3) throughout the optimization ("RL-LR Fixed λ"). This corresponds to a direct application of RL without problem-specific insights used in our RL-LR method. The fixed-λ overhead has negative savings at 0.79ns and cannot meet the tight target specification of 0.77ns. Thus, fixed user-defined λ_i weights are suboptimal as they predetermine the effort that the optimizer makes towards meeting each term in the objective function. In contrast, RL-LR ECO training and Inference flows use an R-GCN agent that *dynamically* updates the λ_i values; both outperform the fixed-λ case. The success of our RL-LR methods is achieved by running an RL-based solution of the constrained optimization problem, with automated tuning of the λ_is that weight the constraint functions relative to the objective.

4.5.4 Transferability across designs

To train over a transferable model across multiple designs, we select four designs highlighted in blue in Table 4.5 and perform offline RL training. The table shows that in thirteen testcases, inference (Inference across Designs) using the trained model can achieve the target delay; "–" represents an unachievable target delay. The model works for all testcases under low ($\leq$5mV) IR drop, and all but one in the moderate ($\leq$10mV) IR drop regime, as the timing degradation for the former is smaller and easier to fix. We do not consider high IR drops that would require major changes rather than ECOs.

Since training across designs builds a "common" model for the four designs, the zero-shot model may fail to meet the target delay of unseen testcases (circuits that are not blue). If so, we apply inexpensive fine-tuning on the trained model individually for each testcase, and can then achieve the target timing in all cases. We almost always have lower power cost than the LR baseline, and much lower runtime than training from

Table 4.5: Transferability across designs

IR Drop	Designs	Target Delay (ns)	Initial Delay (ns)	% Power Overhead Savings			Runtime(s)	
				LR Baseline	Inf-D	Fine Tuning	LR Baseline	Inf-D + Fine Tuning
Low (≤5mV)	des_area	0.56	0.568	76.4%	78.7%	NA	55	24
	wb_dma	0.47	0.475	20.6%	21.0%	NA	39	12
	pid_controller	0.75	0.764	51.6%	51.7%	NA	54	20
	aes_cipher_top	0.89	0.908	4.3%	16.7%	NA	213	121
	des_perf	0.59	0.609	14.3%	18.4%	NA	514	142
	pci_bridge32	0.75	0.772	24.2%	24.5%	NA	185	108
	wb_conmax	0.77	0.829	21.6%	23.8%	NA	274	120
Moderate (≤10mV)	des_area	0.56	0.590	35.2%	45.7%	NA	82	36
	wb_dma	0.47	0.486	16.7%	19.1%	NA	47	33
	pid_controller	0.75	0.781	32.6%	33.7%	NA	58	19
	aes_cipher_top	0.89	0.941	8.6%	29.8%	NA	654	94
	des_perf	0.60	0.632	8.3%	11.3%	NA	576	175
	pci_bridge32	0.75	0.781	16.0%	16.2%	NA	92	15
	wb_conmax	0.77	0.834	11.8%	-	16.5%	290	37+294=331
Average				24.4%	30.0%			–55.7%
Maximum				76.4%	78.7%			–85.6%

scratch. Designs that successfully meet their target periods do not need fine tuning, as shown by "NA." Our runtime improvement is 55.7% over LR on average. *In 13 of 14 testcases, fine-tuning is not needed, and our runtime is better than that for LR* (one testcase is tied). For wb_conmax under moderate IR drop, fine tuning shows power improvement. The runtime column here shows the sum of Inf-D and fine-tuning runtimes.

4.6 Conclusion

Our RL-LR ECO method recovers degradations to the power-delay tradeoff curve due to IR drops. We incorporate problem-specific knowledge into RL-driven gate sizing. Together with other problem-specific methods, our method shifts the Pareto optimal front of the power-delay tradeoff curve to the left. Our methods range from a full training flow per design to zero-shot inference on unseen specifications, as well as a fine tuning flow on a pretrained model. We believe that our LR-based weight update strategy is generalizable for solving any constrained optimization problem using RL.

Chapter 5

ML-based AIG Timing Prediction to Enhance Logic Optimization

5.1 Introduction

As technology scales down to smaller nodes, logic optimization has become more critical than ever before. Suboptimal logic structures can lead to increased delay, area, and excessive power consumption. Several techniques for logic structure optimization have been extensively studied in previous works. Heuristic methods [62] may converge quickly, but could get stuck in local optima; SAT-based methods [63] [64] perform an exhaustive search to find optimal solutions but are impractical for optimizing large designs due their high computational cost; simulated annealing (SA) [65, 66] is effective in exploring the global solution space but may be computationally expensive; genetic algorithms [67], based on evolutionary approaches, can be effective for complex problems, but require more parameters and tuning effort than SA.

In recent years, machine learning (ML) techniques have been applied in logic optimization with promising results. In [68], a decision tree-based approach has been applied to minimize the depth and the number of nodes in an and-inverter graph (AIG) logic representation. Neural networks have been employed in [69] to choose appropriate logic optimizers for different parts of a circuit for modern designs that consist of heterogeneous blocks. Other works include [70], which uses long short-term memory (LSTM) networks to predict design metrics for a given synthesis flow applied on a design, and [71], which focuses on predicting timing for a given optimization sequence,

but under a limited range of logic transformations. A self-evolving system for synthesis parameter tuning [72] is developed to automate solution space exploration. Logic optimization has been formulated as a reinforcement learning (RL) problem, as in [73] (for majority-inverter graphs (MIGs)) and [74] (for AIGs), where RL has been explored for learning a better sequence of transformations to improve the quality of the design. In [73], only the depth of MIG is optimized in the proposed flow, while in [74], the search space only consists of seven primitive transformations and the reward function only considers the direction of the delay and area change instead of quantifying how much change each action brings.

A major limitation of these prior approaches is that they utilize proxy metrics, e.g., taking the delay to be proportional to the logical depth of the circuit, and incorporate them into objective functions for optimization. However, proxy delay metrics may not be well correlated with the post-synthesis delay after technology mapping, and can be ineffective in optimizing the design. To overcome these limitations of prior methods, this work studies how logic optimization can be improved by incorporating alternatives to approximate proxy metrics, to compute the delay in the cost function of a conventional optimization paradigm. These performance metrics must be computed efficiently to maintain the ability of logic synthesis to explore the large solution space efficiently, so that synthesis is scalable to large designs.

In principle, it is possible to integrate the post-mapping delay in the cost function by performing the full technology mapping step. Our study finds that this can lead to a design of better quality than the optimization based on proxy metrics, but due to the high cost of the mapping step, the procedure is not scalable to large designs. Therefore, we propose an ML-enhanced timing-aware optimization approach to speed up logic optimization by building a predictor for the circuit delay after technology mapping and timing analysis. This approach is shown to be fast and capable of achieving optimization results of similar quality as the ground truth based on post-mapping delay. Our work makes the following contributions:

1. We show that the correlation between the proxy of maximum delay and post-mapping maximum delay is poor.

2. We demonstrate that a logic optimization flow driven by post-mapping delay leads to better outcomes than using proxy metrics, but the runtime can be 20× larger.

3. We propose an ML-enhanced optimization flow that significantly speeds up the

logic optimization driven by post-mapping delay, while maintaining the quality of solution, by incorporating ML inference in the cost calculation.

5.2 Background

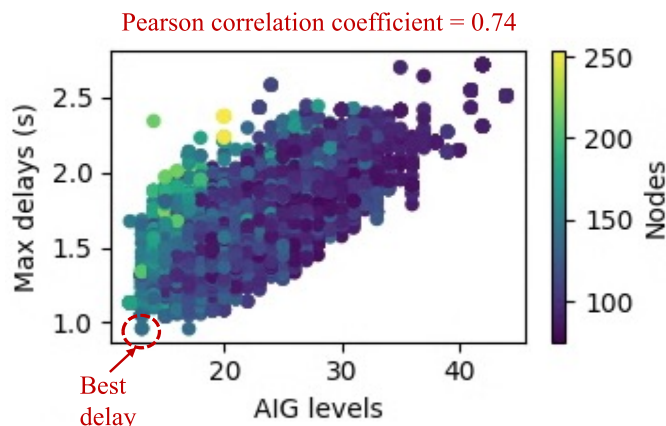


Figure 5.1: Scatter plot: post-mapping circuit delay vs. the number of AIG levels.

Logic synthesis is a process that turns an abstract specification of desired circuit behavior into a design implementation in terms of logic gates. It involves logic optimization, technology mapping, and post-mapping optimization. The AIG, which decomposes the netlist into an elemental network of AND and Inverter functions, is one of most widely-used initial representations of a netlist, and facilitates the application of structural optimizations to the circuit. The goal of AIG optimization is to apply a sequence of transformations to optimize the power performance and area (PPA) of the circuit, while maintaining the equivalence of the logic to the original specification. Intermediate AIG optimization steps often rely on proxy metrics to estimate final timing and area of the circuit: specifically, the node count of the AIG is used to approximate the design area, and the number of logic levels in the AIG is used to estimate maximum delay. However, the PPA after logic synthesis, determined by technology mapping and timing analysis on gate-level netlist, can be inconsistent with the PPA predicted by these proxy metrics. Thus, the design may not be effectively optimized based on inaccurate PPA estimates.

Table 5.1: Post-mapping performance for two AIGs with the same number of levels and nodes

AIG Candidate	**Level**	**Node Count**	**Post-mapping Delay (ns)**	**Post-mapping Area (um^2)**
AIG1	14	178	1.75	803.27
AIG2			1.33	770.74

5.2.1 Limitations of optimizing proxy metrics

As mentioned above, the correlation between number of AIG levels and the corresponding maximum delay after mapping is imperfect. Fig. 5.1 plots the post-technology-mapping maximum delay against the number of levels, for a number of AIGs associated with a multiplier design. From this experiment, we find that the correlation between maximum delay and the number of AIG levels is only 0.74. Moreover, as seen in the figure, the best delay after mapping does not correspond to an AIG with the smallest number of levels. Moreover, another design with fewer AIG levels that the optimal design has $> 1.5\times$ larger delay. These observations indicate that proxy metrics may not accurately guide the AIG optimization to achieve the "true" optimal design with the best performance.

These inaccuracies expose another major limitation in the use of AIG-optimization-based proxy metrics: two AIGs with same node count and level may have significantly different delay and area at the post-mapping stage. Table 5.1 shows these metrics for two representations, AIG1 and AIG2, for the same circuit. An optimizer that uses the number of levels as a proxy for delay would treat them as the same and randomly choose one to continue the optimization. This may cause the search to miss a good candidate and thus lose the opportunity to fully explore the design space.

5.2.2 Performance-driven logic optimization

To resolve the aforementioned issues, we study a performance-driven logic optimization flow. Instead of relying on proxy metrics, the exact post-mapping delay and area are used directly in the cost function to guide AIG optimization, aiming to achieve a more precise and effective optimization process. To demonstrate the effectiveness of introducing the post-mapping delay and area, we compare the outcomes of two different optimization flows: the *baseline flow*, which drives logic optimization using proxy metrics in the cost function, and a *ground-truth-based flow*, which uses the post-timing

delay and area instead of proxy metrics. For a multiplier design, we conducted a hyperparameter sweep for these two logic optimization flows to optimize the AIG. We find that for two AIGs with the same area but from the Pareto-optimal fronts of the two flows, the delay of the AIGs optimized using the ground-truth-based flow can be up to 22.7% better than those optimized using the baseline flow. This suggests that using exact post-mapping delay and area in the optimization process can lead to a more comprehensive exploration of the design space and result in better delay and area.

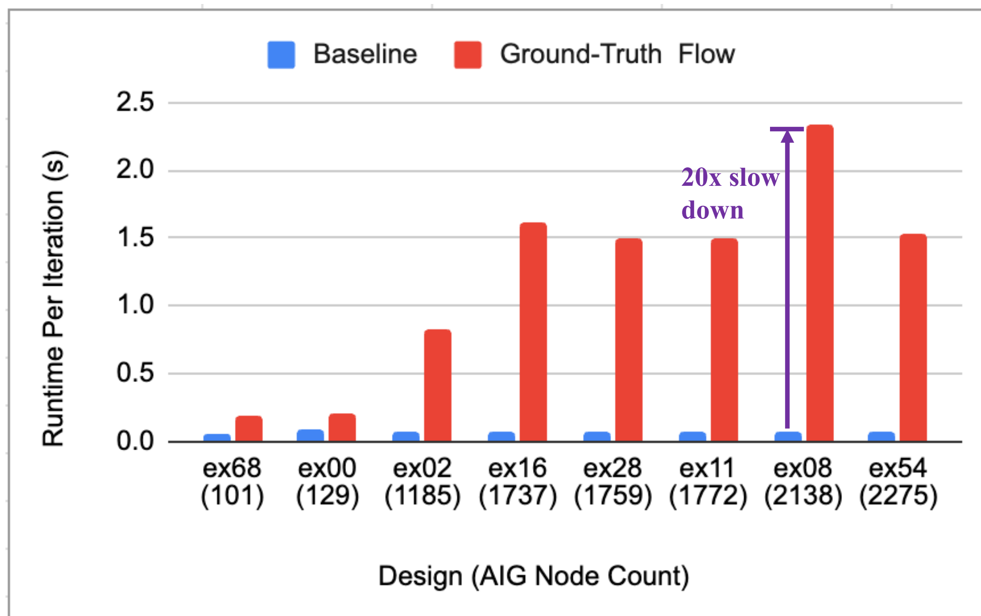


Figure 5.2: Runtime comparison for one iteration in the original logic optimization flow and the ground-truth-based logic optimization flow. The x-axis shows the name of each designs, with the number of its AIG nodes in parentheses.

However, ground-truth-based logic optimization requires technology mapping and STA runs during each iteration of the logic optimization flow, which is computationally intensive. Such a flow is found to be up to 20× slower, on eight designs from IWLS benchmarks [75], as illustrated in Fig. 5.2. While the runtimes per iteration are relatively modest, it should be noted that emerging high-quality synthesis approaches, e.g., [66] may require tens and thousands of iterations. For small-scale circuits, such as ex00 and ex68, even for this large number of iterations the runtime for this ground-truth-based flow may be affordable, and the original flow could be replaced by a ground-truth-based flow to achieve much better PPA. However, for larger designs, the runtime for

ground-truth-based optimization becomes prohibitively long.

Thus, it is essential to find an alternative to computing the ground truth delay by running the mapping and static timing analysis (STA) steps. We propose to build an ML model to predict post-mapping delay, thereby avoiding the need to run mapping and STA in every iteration. This is used to develop an ML-enhanced logic optimization flow in this work, with the aims to achieve AIGs with

comparable quality metrics as ground-truth-based optimization, but with a substantial reduction in runtime.

5.3 Methodology

5.3.1 ML-enhanced logic optimization

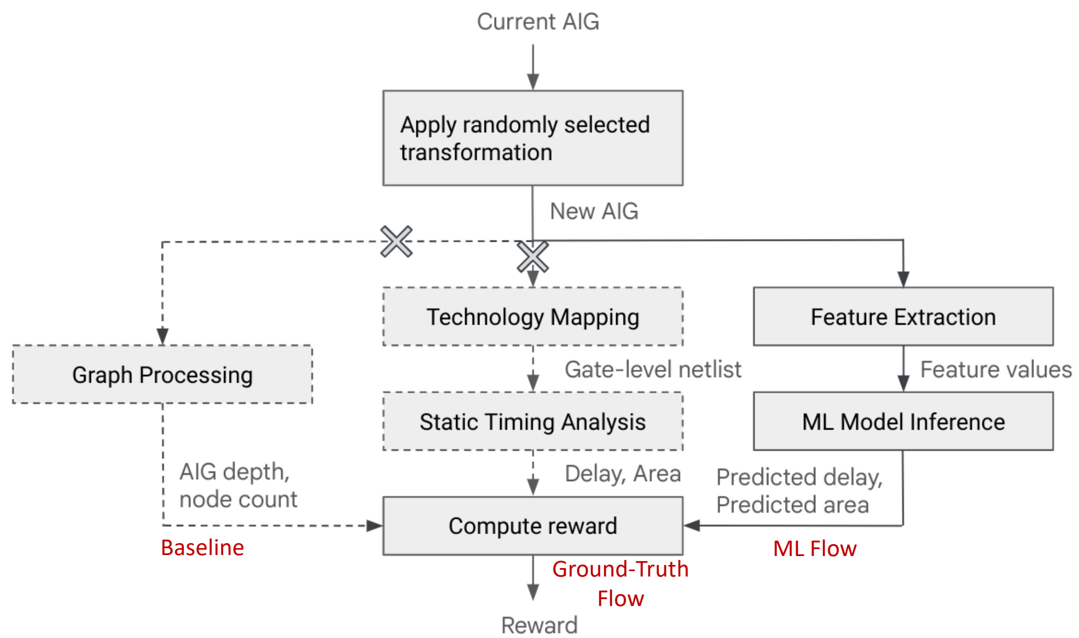


Figure 5.3: Three flows for AIG optimization.

Three logic optimization flows discussed in Sec.5.2 are illustrated in Fig. 5.3. From left to right in the figure:
(1) The **baseline flow** is the original flow with a graph processing step for AIG optimization in which a randomly selected transformation is applied to the AIG at each

iteration to produce a new AIG. An industry flow that we are familiar with uses 103 combinations of the basic transformations [76] available in ABC [77], from which one combination is selected in each iteration and applied to the AIG. The depth and the node count of the new AIG are obtained by processing the graph to evaluate the delay and area in the reward calculation, which determines whether the new AIG will be accepted as the start point for the exploration in the next iteration.
(2) The **ground-truth-based flow** performs the full technology mapping step, followed by STA, for each new AIG to obtain the post-mapping delay of the mapped gate-level netlist. These precise metrics are used to compute the cost and guide the optimization process. While this method provides accurate PPA results, it is computationally expensive due to the high cost of repeated mapping and STA, and is impractical for large designs.
(3) Our **proposed ML-based flow** leverages ML models to predict post-mapping delay using features extracted from an AIG. Instead of running mapping and STA in each iteration, a pretrained ML model estimates the delay to guide the optimization, which aims to speed up the flow with high prediction accuracy. A large design, where the ground-truth-based flow incurs large runtimes, can greatly benefit from this flow.

5.3.2 Feature engineering

We now enumerate the features used in our approach to drive a decision tree ML model. In principle, it is possible to use graph neural networks (GNNs), which are good at analyzing complex data. However, as the number of features for each node in AIG is limited, GNNs are unable to outperform decision-tree-based models on predicting the maximum delay of an AIG. Our experiments find that not only is the GNN-based timing prediction is 2% worse than decision-tree-based model on average across the designs used in our experiment, but the training cost is also much higher than the lightweight decision-tree-based model. In the context of AIGs, the features available at each node, such as fanin, fanout, or logic type, are relatively simple, and do not fully leverage the strengths of GNNs, and this explains their inability to deliver better performance than decision tree models. Moreover, the maximum delay of a graph are often dominated by several long path and is not greatly affected by the remaining paths, which is hard for a GNN to learn based on its paradigm of message passing. Thus, we choose decision-tree-based model and extract graph-level features, which is more efficient in learning than relying on node-to-node interactions in our case, for model training.

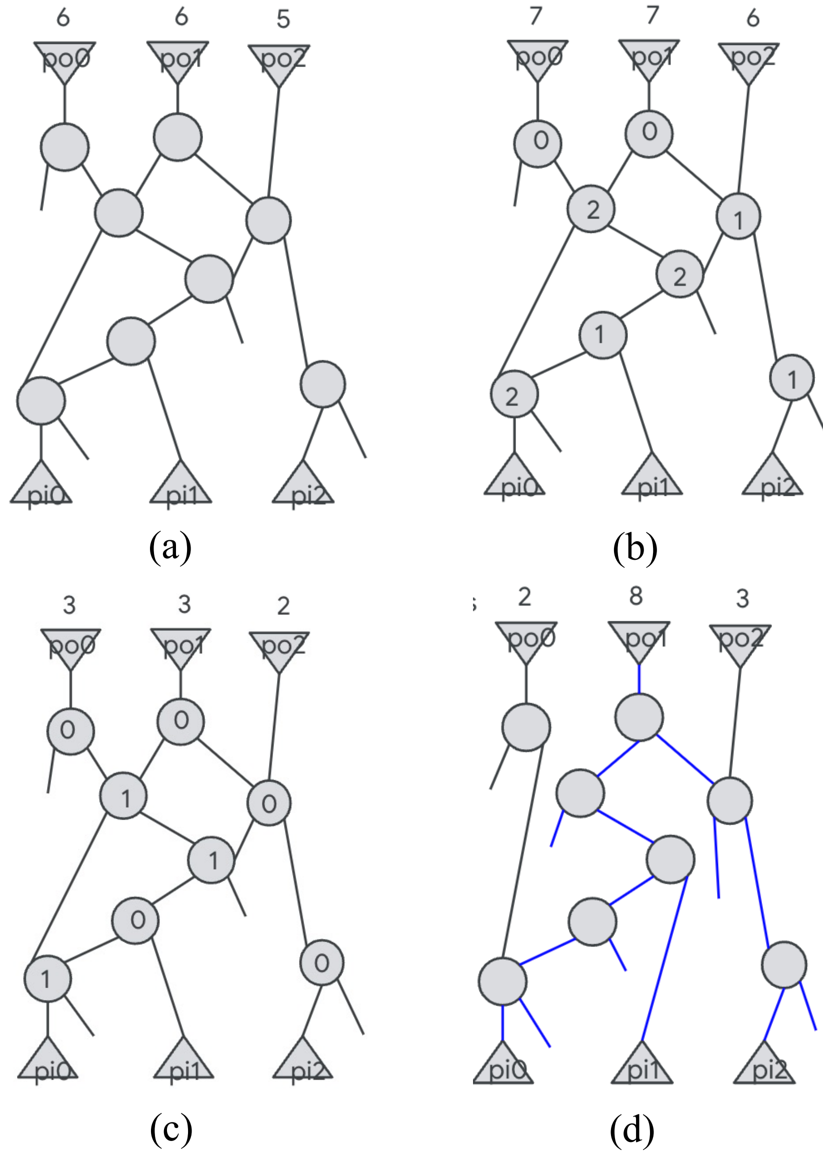


Figure 5.4: Feature extraction in an example AIG. (a) The maximum depth is annotated for each PO. (b) The fanout of each node is annotated as the weight and the updated maximum depth is annotated for each PO. (c) The binary-encoded fanout of each node is annotated as the weight and the updated maximum depth is annotated for each PO. (d) The subgraph of PO1 is highlighed in blue, and the number of paths is annotated for each PO.

The features extracted for each AIG are summarized in Table 5.2, and are based on an analysis of the sources of timing miscorrelation between AIG depth and the maximum delay of the mapped netlist. Two primary sources of this miscorrelation are considered in this work: (a) the path depth change in the mapped netlist compared to the AIG structure, which impacts the number of stages in the critical path, thus affecting the overall path delay; and (b) the fanout change after mapping, in which several nodes in AIG are merged into a large cell, resulting in changes to the fanout of the node driving these cells, which affects the gate delay and contributes to the miscorrelation. The features extracted from AIG in this work can be classified into three categories: the features associated with the critical path, the features related to the fanout distribution, and the features related to the structural complexity of the subgraph correponding to each primary output (PO) reflecting how the topology may affect timing after mapping. Fig. 5.4 demonstrates how the features are extracted using an example AIG with paths between three primary inputs (PIs) and three POs. Different types of depth calculation for PO are shown in Fig. 5.4(a)-(c), which take the nodes between PO and PI, including the node for PI and excluding the PO node (which is a gate output). Fig. 5.4(d) shows an example of subgraph extraction for PO.

The features associated with the critical path include three types of features related to AIG depth.

aig_nth_long_path_depth: For each PO, the largest depth is obtained by traversing the graph. Fig. 5.4(a) annotates each PO by listing the maximum depth from a PI. In our experiments, we take the top n maximum depths among all POs as our features. The value for parameter n is included in the “Comments” column of Table 5.2.

aig_nth_weighted_path_depth: The feature takes the impact of fanout into account on potential critical paths. The fanout of each node is annotated as the weight of the node, and the graph is traversed to compute the largest depth for each PO with these weights, i.e., with the fanout considered. The top n weighted depths are then used as features.

aig_nth_binary_weighted_path_depth: The feature considers the probability of nodes being merged into large cells during mapping, which can reduce the length of some paths and reduce delay consequently. Nodes with high fanout have a lower probability of being merged into a large cell. In our approach, nodes with two or more fanouts are assigned a weight of 1, while nodes with fewer than two fanouts are given a weight of 0. Then the largest depth for each PO is updated. The binary weights help model the potential reduction in path depth after mapping. In our experiments, we empicically set the

Table 5.2: Features extracted from the AIG

AIG features	**Comments**
number_of_node	Number of nodes in AIG
aig_level	Level of AIG graph
aig_nth_long_path_depth	The nth max depth of all POs ($n = 1, 2, 3$ in experiments)
aig_nth_weighted_path_depth	The nth max depth weighted by node fanout of all POs ($n = 1, 2, 3$ in experiments)
aig_nth_binary_weighted_path_depth	The nth max depth weighted by 0 (when the node inputs 2), 1 (when the node inputs = 2) of all POs ($n = 1, 2, 3$ in experiments)
fanout_mean, max, std, sum	Mean, max, standard deviation, and sum of the fanout of all nodes
long_path_fanout_mean, max, std, sum	Mean, max, standard deviation, and sum of fanout of nodes on long path (path_depth = aig_level)
num_of_paths	Number of paths of each PO (choose top n largest)

parameter n in this path-depth-related feature to 3.
fanout_mean, max, std, sum: For the features related to fanout, high-fanout nodes are likely to have high load capacitance, which results in large gate delay. Therefore, we include these features related to the fanout distribution over the graph. These features quantify the overall fanout distribution throughout the AIG by computing the mean, maximum, standard deviation, and sum of fanouts for all nodes. If fanout is uneven across the netlist, with certain regions having much higher fanout than others, the paths with higher fanout are more likely to dominate the overall delay.
long_path_fanout_mean, max, std, sum: In a similar way, we gain insights into how fanout distributes at different stages over a long path. If fanout is unevenly distributed along the path, the large delays at the high-fanout stages will result in large path delay due to their higher load capacitance.

To capture the complexity of the paths to each PO, the number of paths is considered by traversing the subgraph of each PO.
number_of_paths: This feature approximates the probability of a PO having multiple critical and near-critical paths and avoids explicitly enumerating all near-critical paths which is a computationally expensive step for feature extraction.

5.3.3 Data generation and model training

We collect data from eight IWLS benchmarks [75] for both training and testing. For each design, we generate 40,000 unique AIGs by randomly applying a series of logic transformations available in ABC [77], a widely adopted open-source logic synthesis framework, to build an initial AIG representation of the design. The maximum delay labels are generated by performing technology mapping and STA under a 130nm technology [28], using ABC to map each AIG to a standard cell library.

The ML models are implemented using XGBoost [13], an ensemble learning algorithm based on gradient boosting. The model is trained using root mean squared error (RMSE) as the loss function. The hyperparameter values are chosen based on grid search. For the XGBoost regressor, we use a learning rate 0.01. We choose the maximum tree depth = 16, the number of estimators = 5000, and the subsampling ratio = 0.8.

5.4 Experimental setup and results

Our experiments are conducted on eight designs from the IWLS benchmark suite, each from a different functional category and with more than three POs, minimizing similarity between designs and ensuring diversity of the data. Designs with fewer than three POs tend to be simpler and can be handled efficiently without using ML inference, and are therefore not considered in our experiments. The first three columns in Table 5.3 summarize the design names, the number of PIs and POs, the median number of AIG nodes across 40K generated AIGs for each design, which range from 69 to 2290 nodes; the precise range for the set of AIGs for each benchmark is shown in the third column of the table. Four designs are used for model training, and four designs are used for testing to evaluate the ability of the mode to be generalized to unseen designs. We evaluate the prediction accuracy of the model across all designs and compare the ML-based SA logic optimization flow against both the baseline flow and the ground-truth-based flow. The experiments are conducted based on simulated annealing (SA) paradigm which has been applied for circuit optimization with ML in [66]. Our models can also be integrated into other conventional approaches besides SA. In this work, we choose SA considering two main factors: 1) compared to deterministic algorithms, SA allows to accept temporary cost-increasing solutions with a certain probability during the search process, allowing "hill-climbing" that can enable the optimization to potentially find better solutions later;

Table 5.3: Accuracy of XGBoost model for timing prediction

Design		PI/PO	#Node (Range)	Mean of % Error	Max of %Error	Std. of % Error
Training	**EX00**	16/7	69-189	4.24%	29.87%	3.56%
	EX08	18/5	1828-1448	1.90%	14.05%	1.59%
	EX28	17/7	1296-2222	1.53%	14.41%	1.36%
	EX68	14/7	62-140	4.23%	33.64%	3.48%
Test	**EX02**	18/6	848-1522	5.77%	32.52%	4.86%
	EX11	17/7	1253-2290	5.22%	36.96%	3.90%
	EX16	16/5	1237-2236	4.50%	36.74%	3.55%
	EX54	17/7	1469-3080	4.83%	39.85%	3.87%
				Avg. 4.03%	Max 39.85%	Ave. 3.27%

2) The SA implementation allows the designer to customize methods for estimating the PPA and for tuning the weights for each components in cost function, making it more versatile in handling complex trade-offs. All runtimes are reported on an AMD EPYC 7B13 CPU @2.3GHz.

5.4.1 Evaluation of model accuracy

Table 5.3 summarizes the metrics for the ML model and its prediction accuracy across all designs. The metrics used for evaluation include the mean, maximum, and standard deviation of the absolute %error as metrics for evaluation, where the absolute %error is the absolute difference between the ground-truth and the predicted value with respect to the ground-truth value. The results show that the average prediction error across all designs is 4.03%, demonstrating good overall accuracy, and the average standard deviation is 3.27% across the designs, so the prediction for most graphs are within a small error.

5.4.2 Evaluation of ML-enhanced logic optimization

The three flows described in Section 5.3.1 are applied to the eight IWLS benchmarks and evaluated in terms of both their quality of solution and runtime. Fig. 5.5 shows the optimal AIGs from these three flows. The red dots represent the outcomes from baseline optimization flow, the black dots represent the outcomes from the ground-truth-based flow, and the green dots represent the outcomes from the ML-based flow. Each point on the plot corresponds to an optimal AIG obtained from a specific run of SA logic optimization flow with a certain hyperparameter setting. We sweep the hyperparameters

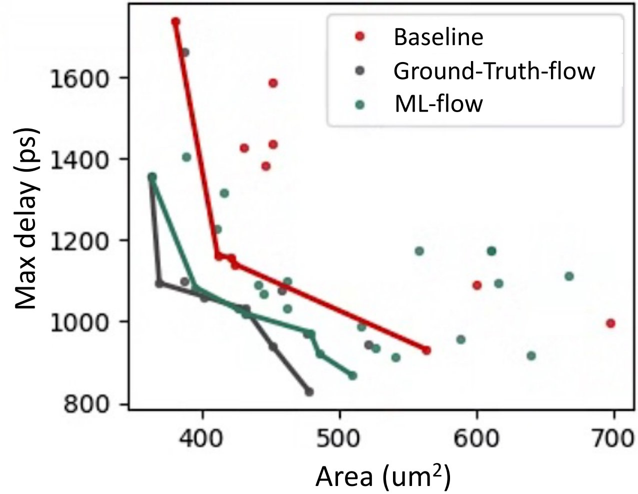


Figure 5.5: A comparison of the Pareto-optimal fronts for the delay and area of a test design from the baseline, ground-truth-based, and ML-based flows.

Table 5.4: Runtime for the three flows

Design		Baseline (s)	Ground-Truth-flow Mapping+STA (s)	ML-flow ML Inference (s)
Training	**EX00**	0.0936	0.3028	0.1118 (-63.08%)
	EX08	0.0736	2.3375	0.1967 (-88.79%)
	EX28	0.0712	1.4966	0.1555 (-85.54%)
	EX68	0.0463	0.1961	0.0166 (-74.05%)
Test	**EX02**	0.0670	0.8335	0.1213 (-79.09%)
	EX11	0.0693	1.5044	0.1575 (-85.59%)
	EX16	0.0764	1.6131	0.1617 (-85.91%)
	EX54	0.0766	1.5320	0.1715 (-84.58%)
Avg.				-80.83%
Max				-88.79%

to obtain an optimal AIG for different flow settings, which involves sweeping relative weights in cost function and sweep the annealing temperature decay rate. For each flow, a Pareto-optimal curve is generated and shown in the figure. The green curve for the ML-based flow is very closed to the black curve for the ground-truth-based flow, demonstrating that the ML-enhanced approach achieves nearly the same level of quality in terms of delay and area. Both the green and black curves are significantly better than the red curve for the baseline optimization flow using conventional proxy metrics, showing that the ML-based and ground-truth-based flows outperform the original approach in exploring and identifying better designs.

Table 5.4 summarizes the runtime of single iteration of the three different flows. The runtime for the baseline flow includes the time elapsed for applying AIG transformation, graph processing to obtain the depth and the node count of the new AIG to evaluate the cost. The runtime for ground-truth-based flow requires a large additional runtime overhead for performing technology mapping and STA on the new AIG. In contrast, the ML-based flow needs a small amount additional runtime for feature extraction and ML inference on the new AIG, but avoids the costly mapping and STA. The ML-based flow achieves a significant runtime reduction of 80.83% compared to the ground-truth-based flow on average and maximum 88.79% reduction across all testcases, while delivering similar quality of solution. This demonstrates that the ML-enhanced logic optimization flow can not only maintain high-quality results but also make substantial improvements on efficiency, which makes it an excellent alternative for larger designs.

5.5 Conclusion

This chapter has proposed an ML-enhanced performance-driven logic optimization flow that addresses the limitations of conventional logic optimization flow relying on proxy metric for delay estimation. While a ground-truth-based flow, incorporating exact post-mapping delay, improves design quality, it significantly increases runtime. Our experimental results show that our approach offers a practical solution for large designs, which generates design of better quality with small runtime overhead.

Chapter 6

Conclusion

This thesis has addressed critical challenges in physical design , focusing on improving QoR prediction accuracy, optimization efficiency, and the quality of the design using ML techniques. We have investigated timing prediction and optimization challenges at the global routing stage in the absence of detailed routing information. ML-based models are proposed to bridge the gap between GR-based and post-DR timing estimations, achieving higher accuracy for timing and parasitic predictions. Experimental evaluations use both open-sourced and commercial tool flows, demonstrating that these models lead to improved DR outcomes, and our approach are computationally efficient. Moreover, our experimental results show that our models generalize well to designs generated using unseen clock periods, and that our flow generates better DR solutions even when training data are noise (Chapter 3). After routing has been completed, ECO optimization is necessary to make the design satisfy the constraints in late design stage without making larger perturbation, which can be challenging for congested designs. We contribute an RL-LR ECO method to recover the degradations caused by IR drops. The problem-specific knowledge has been incorporated into RL-driven gate sizing. Together with other problem-specific methods, our method shifts the Pareto optimal front of the power-delay tradeoff curve to the left. Our methods range from a full training flow per design to zero-shot inference on unseen specifications, as well as a fine tuning flow on a pretrained model (Chapter 4).

Early design stages are also crucial as they establish the foundation for the entire chip design process, directly influencing PPA metrics. Good QoR prediction in early stage ensures efficient exploration of the design space and reduces costly design iterations. We contribute an ML-enhanced performance-driven logic optimization flow that addresses

the limitations of conventional logic optimization flow that rely on proxy metrics for delay estimation in logic synthesis. By incorporating exact post-mapping delay into the optimization process, the proposed approach is shown to improve design quality. To save the runtime overhead of ground-truth-based methods, ML models are employed to predict post-mapping performance metrics efficiently. This offers a practical solution for large-scale designs, which generates design of comparable quality with small runtime overhead (Chapter 5).

In summary, this thesis has contributed multiple novel approaches and demonstrated how ML can be effectively integrated into circuit design methods to enhance QoR prediction accuracy, optimize design quality, and improve design efficiency, which makes a step forward in leveraging ML to tackle complex problems in modern IC design with advanced technology nodes.

References

[1] (2024) The growing challenge of semiconductor design leadership. [Online]. Available: https://www.semiconductors.org/wp-content/uploads/2022/11/2022_The-Growing-Challenge-of-Semiconductor-Design-Leadership_FINAL.pdf

[2] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv:1609.02907*, 2017.

[3] (2024) International roadmap for devices and systems: More moore. [Online]. Available: https://irds.ieee.org/editions/2023/20-roadmap-2023-edition/130-irds%E2%84%A2-2023-more-moore

[4] A. Agnesina, K. Chang, and S. K. Lim, "VLSI placement parameter optimization using deep reinforcement learning," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2020, pp. 1–9.

[5] V. A. Chhabria, W. Jiang, A. B. Kahng, and S. S. Sapatnekar, "A machine learning approach to improving timing consistency between global route and detailed route," *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 1, 2023.

[6] W. Jiang, V. A. Chhabria, and S. S. Sapatnekar, "IR-aware ECO timing optimization using reinforcement learning," in *Proceedings of the ACM/IEEE International Symposium on Machine Learning for CAD*, 2024.

[7] W. Jiang, J. Yan, and S. S. Sapatnekar, "ML-based AIG timing prediction to enhance logic optimization," *arXiv:2412.02268*, 2024.

[8] C. Chu and Y.-C. Wong, "FLUTE: Fast lookup table based rectilinear steiner minimal tree algorithm for VLSI design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 1, pp. 70–83, 2008.

[9] L. Kannan, P. Suaris, and H.-G. Fang, "A methodology and algorithms for post-placement delay optimization," in *Proceedings of the ACM/IEEE Design Automation Conference*, 1994, pp. 327–332.

[10] S. S. Sapatnekar, *Timing.* Boston, MA: Springer, 2004.

[11] P. R. O'Brien and T. L. Savarino, "Modeling the driving-point characteristic of resistive interconnect for accurate delay estimation," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 1989, pp. 512–515.

[12] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, p. 5–32, Oct. 2001.

[13] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2016, p. 785–794.

[14] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," *arXiv:1703.06103*, 2017.

[15] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," *arXiv:1710.10903*, 2018.

[16] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *arXiv:1706.02216*, 2018.

[17] R. Bellman, "On the theory of dynamic programming." *Proceedings of the National Academy of Sciences of the United States of America*, vol. 38, no. 8, pp. 716–719, 1952.

[18] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with deep reinforcement learning," *arXiv:1312.5602*, 2013.

[19] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proceedings of the International Conference on Machine Learning*, 2016, p. 1928–1937.

[20] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: combining improvements in deep reinforcement learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2018.

[21] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, "Sample efficient actor-critic with experience replay," *arXiv:1611.01224*, 2017.

[22] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv:1707.06347*, 2017.

[23] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," *arXiv:1802.09477*, 2018.

[24] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," *arXiv:1801.01290*, 2018.

[25] S. Dolgov, A. Volkov, L. Wang, and B. Xu, "2019 CAD contest: LEF/DEF based global routing," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2019, pp. 1–8.

[26] A. B. Kahng, L. Wang, and B. Xu, "TritonRoute: The open-source detailed router," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 3, pp. 547–559, 2021.

[27] M. Pan, Y. Xu, Y. Zhang, and C. Chu, "FastRoute: An efficient and high-quality global router," *VLSI Design*, vol. 2012, January 2012.

[28] (2024) SkyWater 130nm PDK. [Online]. Available: https://github.com/google/skywater-pdk

[29] (2024) NanGate 45nm FreePDK and cell library. [Online]. Available: https://si2.org/open-cell-library

[30] (2024) Openroad-flow-scripts. [Online]. Available: https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts

[31] T.-B. Chan, A. B. Kahng, and M. Woo, "Revisiting inherent noise floors for interconnect prediction," in *Proceedings of the ACM/IEEE International Workshop on System-Level Interconnect Pathfinding*, 2020.

[32] H.-H. Cheng, I. H.-R. Jiang, and O. Ou, "Fast and accurate wire timing estimation on tree and non-tree net structures," in *Proceedings of the ACM/IEEE Design Automation Conference*, 2020, pp. 1–6.

[33] W.-T. J. Chan, K. Y. Chung, A. B. Kahng, N. D. MacDonald, and S. Nath, "Learning-based prediction of embedded memory timing failures during initial floorplan design," in *Proceedings of the Asia-South Pacific Design Automation Conference*, 2016, pp. 178–185.

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

[35] T. Yang, G. He, and P. Cao, "Pre-routing path delay estimation based on transformer and residual framework," in *Proceedings of the Asia-South Pacific Design Automation Conference*, 2022, pp. 184–189.

[36] X. He, Z. Fu, Y. Wang, C. Liu, and Y. Guo, "Accurate timing prediction at placement stage with look-ahead RC network," in *Proceedings of the ACM/IEEE Design Automation Conference*, 2022.

[37] 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 ACM/IEEE Design Automation Conference*, 2022.

[38] Y. Ye, T. Chen, Y. Gao, H. Yan, B. Yu, and L. Shi, "Fast and accurate wire timing estimation based on graph learning," in *Proceedings of the Design, Automation & Test in Europe*, 2023.

[39] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem, G. Pradipta, S. Reda, M. Saligane, S. S. Sapatnekar, C. Sechen, M. Shalan, W. Swartz, L. Wang, Z. Wang, M. Woo, and B. Xu, "Toward an open-source digital flow: First learnings from the OpenROAD project," in *Proceedings of the ACM/IEEE Design Automation Conference*, 2019, pp. 1–4.

[40] V. A. Chhabria, W. Jiang, A. B. Kahng, and S. S. Sapatnekar, "From global route to detailed route: ML for fast and accurate wire parasitics and timing prediction," in *Proceedings of the ACM/IEEE International Workshop on Machine Learning for CAD*, 2022, p. 7–14.

[41] (2024) Openroad. [Online]. Available: https://github.com/The-OpenROAD-Project/OpenROAD

[42] (2024) OpenSTA. [Online]. Available: https://github.com/The-OpenROAD-Project/OpenSTA

[43] V. A. Chhabria, A. B. Kahng, M. Kim, U. Mallappa, S. S. Sapatnekar, and B. Xu, "Template-based PDN synthesis in floorplan and placement using classifier and CNN techniques," in *Proceedings of the Asia-South Pacific Design Automation Conference*, 2020, pp. 44–49.

[44] J. P. Fishburn and A. E. Dunlop, "TILOS: A posynomial programming approach to transistor sizing," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 1985, pp. 326–328.

[45] J. Hu, A. B. Kahng, S. Kang, M.-C. Kim, and I. L. Markov, "Sensitivity-guided metaheuristics for accurate discrete gate sizing," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2012, pp. 233–239.

[46] K. Kasamsetty, M. Ketkar, and S. Sapatnekar, "A new class of convex functions for delay modeling and its application to the transistor sizing problem," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 7, pp. 779–788, 2000.

[47] C.-P. Chen, C. Chu, and D. Wong, "Fast and exact simultaneous gate and wire sizing by lagrangian relaxation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 7, pp. 1014–1025, 1999.

[48] A. Sharma, D. Chinnery, T. Reimann, S. Bhardwaj, and C. Chu, "Fast lagrangian relaxation-based multithreaded gate sizing using simple timing calibrations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 7, pp. 1456–1469, 2020.

[49] H.-Y. Chang, I. H.-R. Jiang, and Y.-W. Chang, "ECO optimization using metal-configurable gate-array spare cells," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 11, pp. 1722–1733, 2013.

[50] J. Lee and P. Gupta, "ECO cost measurement and incremental gate sizing for late process changes," *ACM Transactions on Design Automation of Electronic Systems*, vol. 18, no. 1, Jan. 2013.

[51] H.-Y. Lin, Y.-C. Fang, S.-T. Liu, J.-X. Chen, C.-M. Li, and E. J.-W. Fang, "Automatic IR-Drop ECO using machine learning," in *Proceedings of the IEEE International Test Conference in Asia*, 2020, pp. 7–12.

[52] H. Wang, K. Wang, J. Yang, L. Shen, N. Sun, H.-S. Lee, and S. Han, "GCN-RL circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning," in *Proceedings of the ACM/IEEE Design Automation Conference*, 2020.

[53] H. Ren, S. Godil, B. Khailany, R. Kirby, H. Liao, S. Nath, J. Raiman, and R. Roy, "Optimizing VLSI implementation with reinforcement learning," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2021.

[54] Y.-C. Lu, S. Nath, V. Khandelwal, and S. K. Lim, "RL-Sizer: VLSI gate sizing for timing optimization using deep reinforcement learning," in *Proceedings of the ACM/IEEE Design Automation Conference*, 2021, pp. 733–738.

[55] X. Zhou, J. Ye, C.-W. Pui, K. Shao, G. Zhang, B. Wang, J. Hao, G. Chen, and P. A. Heng, "Heterogeneous graph neural network-based imitation learning for gate sizing acceleration," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2022.

[56] S. Huang, A. Abdolmaleki, G. Vezzani, P. Brakel, D. J. Mankowitz, M. Neunert, S. Bohez, Y. Tassa, N. Heess, M. Riedmiller, and R. Hadsell, "Constrained multi-objective reinforcement learning framework," in *Proceedings of the 5th Conference on Robot Learning*, 2022, pp. 883–893.

[57] A. Irpan, "Deep reinforcement learning doesn't work yet," https://www.alexirpan.com/2018/02/14/rl-hard.html, 2018.

[58] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with deep reinforcement learning," in *Adv. NeurIPS*, 2013.

[59] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge, UK: Cambridge University Press, 2009.

[60] N. Viswanathan and C. C.-N. Chu, "FastPlace: Efficient analytical placement using cell shifting, iterative local refinement and a hybrid net model," in *Proceedings of the International Symposium on Physical Design*, 2004, p. 26–33.

[61] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," *arXiv:1909.01315*, 2020.

[62] R. K. Brayton, A. L. Sangiovanni-Vincentelli, C. T. McMullen, and G. D. Hachtel, *Logic Minimization Algorithms for VLSI Synthesis.* USA: Kluwer Academic Publishers, 1984.

[63] A. Mishchenko and R. Brayton, "SAT-based complete don't-care computation for network optimization," in *Proceedings of the Design, Automation & Test in Europe*, 2005, pp. 412–417 Vol. 1.

[64] N. Sörensson and N. Een, "MiniSat v1.13 – a SAT solver with conflict-clause minimization," in *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, 2005.

[65] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[66] A. Hillier, G. Rotival, I. Lobov, K. Mahajan, M. Gelmi, N. Vu, O. Temam, S. Guadarrama, and V. Nair, "Learning to Design Efficient Logic Circuits," https://www.iwls.org/iwls2023, 2024.

[67] K. Ohmori and T. Kasai, "Logic synthesis using a genetic algorithm," in *Proceedings of the IEEE International Conference on Intelligent Processing Systems*, vol. 1, 1997, pp. 137–142.

[68] B. A. de Abreu, A. Berndt, I. S. Campos, C. Meinhardt, J. T. Carvalho, M. Grellert, and S. Bampi, "Fast logic optimization using decision trees," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, 2021, pp. 1–5.

[69] W. L. Neto, M. Austin, S. Temple, L. Amaru, X. Tang, and P.-E. Gaillardon, "LSOracle: A logic synthesis framework driven by artificial intelligence: Invited paper," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2019, pp. 1–6.

[70] C. Yu and W. Zhou, "Decision making in synthesis cross technologies using LSTMs and transfer learning," in *Proceedings of the ACM/IEEE International Workshop on Machine Learning for CAD*, 2020, pp. 55–60.

[71] H. Zheng, Z. He, F. Liu, Z. Pei, and B. Yu, "LSTP: A logic synthesis timing predictor," in *Proceedings of the Asia-South Pacific Design Automation Conference*, 2024, pp. 728–733.

[72] M. Ziegler, H.-Y. Liu, G. Gristede, B. Owens, R. Nigaglioni, and L. Carloni, "A synthesis-parameter tuning system for autonomous design-space exploration," in *Proceedings of the Design, Automation & Test in Europe*, 2016, pp. 1148–1151.

[73] W. Haaswijk, E. Collins, B. Seguin, M. Soeken, F. Kaplan, S. Süsstrunk, and G. De Micheli, "Deep learning for logic optimization algorithms," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, 2018, pp. 1–4.

[74] A. Hosny, S. Hashemi, M. Shalan, and S. Reda, "DRiLLS: Deep reinforcement learning for logic synthesis," in *Proceedings of the Asia-South Pacific Design Automation Conference*, 2020, p. 581–586.

[75] "IWLS 2024 benchmarks," https://github.com/alanminko/iwls2024-ls-contest/tree/main/benchmarks, 2024.

[76] "ABC commands," https://people.eecs.berkeley.edu/~alanmi/abc/abc.rc, 2024.

[77] R. K. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Proceedings of the International Conference on Computer Aided Verification*, 2010.