iRw: An Intelligent Rewriting

Haisheng Zheng[1], Haoyuan Wu[2], Zhuolun He[2], Yuzhe Ma[3], Bei Yu[2]
[1]Shanghai Artificial Intelligence Laboratory [2]CUHK [3]HKUST (GZ)

***Abstract*—This paper proposes a novel machine learning-driven rewriting algorithm to optimize And-Inverter Graphs (AIGs) for refining combinational logic prior to technology mapping. The algorithm, called iRw, iteratively extracts subcircuits in AIGs and replaces them with more streamlined implementations. These subcircuits are identified using an original extraction algorithm, while the compact implementations are produced through rewriting techniques guided by a machine learning model. This approach efficiently enables the generation of logically equivalent subcircuits with minimal overhead. Experiments on benchmark circuits indicate that the proposed methodology outperforms state-of-the-art AIG rewriting techniques in both quality and runtime.**

I. Introduction

Rewriting is a key technology-independent optimization in logic synthesis. It involves extracting subcircuits, transforming each into a functionally equivalent one, and replacing the original with the transformed version if it offers a lower cost. Rewriting typically targets *single-output subcircuits*, employing various subcircuit extraction methods (e.g., cut enumeration [1], maximum fanout-free cones [2]) tailored to the specific needs of rewriting algorithms. To facilitate better circuit transformations, algorithms such as *pre-computed libraries* [1], [3], *heuristic resynthesis* [2], [4], and *exact resynthesis* [5], [6] have been proposed. Additionally, rewriting with *multi-output subcircuits* [4], [7] enables shared logic exploration across outputs, reducing redundancy and improving performance. Furthermore, to enhance the efficiency of logic optimization, several studies [8]–[10] employ *machine learning* to guide the optimization process.

Recognizing that an effective logic optimization algorithm relies on the subcircuit extraction method and optimization strategies working in tandem, this paper presents several techniques to enhance the rewriting process. The main contributions of this work are summarized as follows:

- A novel subcircuit extraction algorithm specifically designed to extract subcircuits with optimization potential.
- A method of machine learning-guided optimization process, ensuring efficient optimization with minimal runtime.
- Extensive experiments demonstrating that iRw outperforms state-of-the-art AIG rewriting in both quality and runtime.

II. Algorithms

The overall flow of iRw is shown in Fig. 1. iRw iteratively processes each node in the AIG, excluding primary inputs, as the pivot node P through the following stages:

❶ Subcircuit Extraction. A subgraph extraction algorithm is designed to extract subcircuits with optimization potential.

Extract Single-Output Subcircuits. Single-output subcircuits rooted at the pivot node are extracted based on its transitive fan-in, with a maximum input size constraint K. Multiple subcircuits may be extracted for a given pivot node due to this constraint. The breadth-first search algorithm is employed for its layer-wise structure, ensuring the subcircuits cover a broad range of relevant nodes associated with the pivot node.

This work is supported by Shanghai Artificial Intelligence Laboratory.

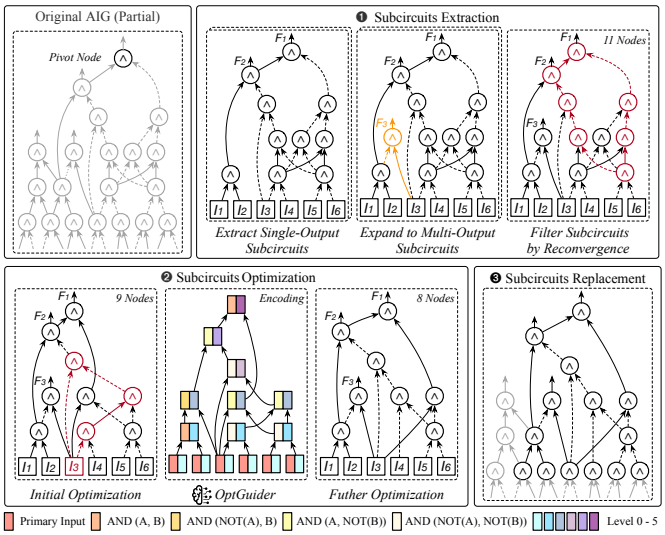

Fig. 1 Overview of the iRw process flow.

Expand to Multi-Output Subcircuits. These single-output subcircuits are then expanded into multi-output subcircuits by exploring their transitive fan-out.

Filter Subcircuits by Reconvergence. Optimization opportunities are assessed based on the presence of *reconvergent paths* within multi-output subcircuits, which indicate potential for observability-based optimizations [4] (e.g., resubstitution [2]). Subcircuits with reconvergent paths are retained for further optimization, while those without are discarded.

❷ Subcircuit Optimization. This stage involves an initial optimization of the extracted subcircuits, followed by an assessment using a lightweight graph neural network (GNN), OptGuider, to determine whether the further optimization stage is needed.

Initial Optimization. *Balance* [1] is employed as the initial optimization technique for the extracted subcircuits due to its negligible runtime overhead and effectiveness in optimizing subcircuits with simple topological structures containing *reconvergent paths*. If reconvergent paths remain after *balance*, *heuristic resubstitution* [4] is then applied, which efficiently optimizes subcircuits with such paths.

Further Optimization. Despite applying *heuristic resubstitution* to the extracted subcircuits, several still retain *reconvergent paths*, indicating potential for further optimization. In these cases, *rewrite* is applied. Even if *rewrite* [1] does not directly improve the circuit, it may generate a cost-equivalent one that

978-3-9826741-0-0/DATE25/© 2025 EDAA

TABLE I Performance Comparison with SOTA Rewriting.

Benchmark		Window Rewriting [4]		iRw			
		Until Convergence		First Iteration		Until Convergence	
Name	# Nodes	# Nodes	Time	# Nodes	Time	# Nodes	Time
adder	1,020	892	0.04	892	0.13	892	0.28
bar	3,336	2,952	3.76	2,952	0.62	2,952	1.17
hyp	214,335	204,926	20.28	204,926	19.90	204,926	40.38
i2c	1,342	1,291	0.10	1,289	0.14	1,273	0.58
int2float	260	239	0.02	232	0.09	226	0.36
log2	32,060	29,700	6.59	29,603	5.19	29,556	28.55
multiplier	27,062	24,566	3.89	24,426	3.93	24,426	8.80
sin	5,416	5,132	1.85	5,115	1.05	5,095	7.73
sqrt	24,618	18,325	2.95	18,279	2.42	18,236	16.17
square	18,484	16,606	2.72	16,386	2.04	16,316	6.47
Average	-	9.79%	-	10.32%	-	10.78%	-
Total	-	-	42.20		35.51	-	110.49

TABLE II Impact of Multi-Output Subcircuit Extraction.

Method	# Initial Nodes	# Optimized Nodes	Average Improvement
b; rs -K 6; rw; rf;	327,933	314,793	4.01%
iRw		304,100	10.32%

TABLE III Impact of OptGuider.

Method	# Nodes [1st Iter.]	Time	# Nodes [Util Conv.]	Time
iRw w/o OptGuider	304,007	44.20	303,805	188.05
iRw with OptGuider	304,100	35.51	303,898	110.49
Improvement	-0.03%	19.66%	-0.03%	41.25%

simplifies *reconvergent paths*, thereby enhancing the effectiveness of the subsequent strategy, *refactor* [1].

Optimization Guider. Our observations reveal that only a few subcircuits benefit from the *further optimization* stage, and reconvergent paths in optimized subcircuits are unreliable indicators for further optimization. However, optimizing these subcircuits reduces both node count in AIGs and optimization iterations. To address this, we propose OptGuider, a lightweight GNN-based algorithm that identifies subcircuits suitable for further optimization, thereby reducing the runtime overhead of iRw. This approach is inspired by several studies [11]–[13] that demonstrate the effectiveness of GNNs in learning netlist representations for downstream tasks.

Node Feature Encoding. As visualized in Fig. 1, instead of marking `NOT` operations on edges, which increases GNN overhead, edge information is incorporated into `AND` gate representations. Node levels, representing the logical hierarchy of AIGs, are captured using positional encoding (PE), improving the model's understanding of circuit structure: $PE_{(v,2i)} = \sin\left(\frac{level(v)}{10000^{2i/d}}\right), PE_{(v,2i+1)} = \cos\left(\frac{level(v)}{10000^{2i/d}}\right)$, where $level(v)$ denotes the logical level of node v, and d represents the feature dimension.

Lightweight GNN. GraphSAGE [14] is used as the GNN model due to its computational efficiency.

Cost-Sensitive Learning. Misclassifying non-optimizable subcircuits increases computational overhead without enhancing the optimization process, while overlooking optimizable subcircuits can impede it. Given this asymmetry, cost-sensitive learning [15] using binary cross-entropy loss is employed in the OptGuider training process to help the classifier prioritize the identification of subcircuits that impact overall performance.

❸ **Subcircuit Replacement.** If the transformed subcircuit shows improvement, it replaces the original one in the AIG.

III. Experiments

iRw was implemented in C++ using mockturtle [16] and LibTorch [17]. Extensive experiments evaluated its performance on a 2.6 GHz AMD EPYC 7H12 CPU. To ensure fairness, the input size for subcircuit extraction was set to $K = 6$ in all configurable algorithms. The circuits from the EPFL benchmark [18], shown in TABLE I, were used for evaluation. The correctness of the optimized AIGs was verified using the combinational equivalence checker from ABC [19].

Comparison with SOTA Rewriting. *Window Rewriting* [4] was used as the baseline. The results are detailed in TABLE I. iRw achieved greater node reduction than the baseline in a single iteration, reducing runtime by `15.83%`. When iRw ran to convergence, the quality improved by `0.99%` on average.

Effectiveness of Multi-Output Extraction. To evaluate the contribution of multi-output subcircuit extraction in iRw, identical optimization strategies were applied to benchmark circuits with and without this extraction. The results in TABLE II demonstrate the effectiveness of multi-output subcircuit extraction in enhancing node reduction in iRw.

Effectiveness of OptGuider. To evaluate the effect of OptGuider on iRw, node reduction performance was compared with and without its guidance. The results in TABLE III indicate that iRw, when guided by OptGuider, achieves significant runtime improvements with only a slight impact on node reduction.

References

[1] A. Mishchenko, S. Chatterjee *et al.*, "DAG-Aware AIG Rewriting: A Fresh Look at Combinational Logic Synthesis," in *Proc. DAC*, 2006.
[2] A. Mishchenko and R. Brayton, "Scalable Logic Synthesis using a Simple Circuit Structure," in *Proc. IWLS*, 2006.
[3] M. Soeken, L. G. Amarù *et al.*, "Optimizing Majority-Inverter Graphs With Functional Hashing," in *Proc. DATE*, 2016.
[4] H. Riener *et al.*, "Boolean Rewriting Strikes Back: Reconvergence-Driven Windowing Meets Resynthesis," in *Proc. ASPDAC*, 2022.
[5] H. Riener *et al.*, "On-the-fly and DAG-aware: Rewriting Boolean Networks with Exact Synthesis," in *Proc. DATE*, 2019.
[6] H. Riener *et al.*, "Exact DAG-Aware Rewriting," in *Proc. DATE*, 2020.
[7] X. Zhu *et al.*, "A Database Dependent Framework for K-Input Maximum Fanout-Free Window Rewriting," in *Proc. DAC*, 2023.
[8] W. L. Neto, M. Austin *et al.*, "LSOracle: A Logic Synthesis Framework Driven by Artificial Intelligence," in *Proc. ICCAD*, 2019.
[9] X. Li, L. Chen, J. Zhang, S. Wen *et al.*, "EffiSyn: Efficient Logic Synthesis with Dynamic Scoring and Pruning," in *Proc. ICCAD*, 2023.
[10] A. B. Chowdhury, B. Tan, R. Carey *et al.*, "Bulls-Eye: Active Few-Shot Learning Guided Logic Synthesis," *IEEE TCAD*, 2022.
[11] H. Zheng, Z. He, F. Liu, Z. Pei, and B. Yu, "LSTP: A Logic Synthesis Timing Predictor," in *Proc. ASPDAC*, 2024.
[12] Z. Shi, H. Pan, S. Khan, M. Li *et al.*, "DeepGate2: Functionality-Aware Circuit Representation Learning," in *Proc. ICCAD*, 2023.
[13] Z. He, Z. Wang, C. Bai, H. Yang, and B. Yu, "Graph Learning-Based Arithmetic Block Identification," in *Proc. ICCAD*, 2021.
[14] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive Representation Learning on Large Graphs," in *Proc. NIPS*, 2017.
[15] C. Elkan, "The Foundations of Cost-Sensitive Learning," in *Proc. IJCAI*, 2001.
[16] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli *et al.*, "The EPFL Logic Synthesis Libraries," in *arXiv preprint*, 2018.
[17] A. Paszke, S. Gross, F. Massa *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Proc. NIPS*, 2019.
[18] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The EPFL Combinational Benchmark Suite," in *Proc. IWLS*, 2015.
[19] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool," in *Proc. CAV*, 2010.