RTLRewriter: Methodologies for Large Models aided RTL Code Optimization

Xufeng Yao[1], Yiwen Wang[2], Xing Li[2], Yingzhao Lian[2], Ran Chen[2], Lei Chen[2], Mingxuan Yuan[2], Hong Xu[1], Bei Yu[1]
[1]Chinese University of Hong Kong [2] Huawei

ABSTRACT

Register Transfer Level (RTL) code optimization is crucial for enhancing the efficiency and performance of digital circuits during early synthesis stages. Currently, optimization relies heavily on manual efforts by skilled engineers, often requiring multiple iterations based on synthesis feedback. In contrast, existing compiler-based methods fall short in addressing complex designs. This paper introduces RTLRewriter, an innovative framework that leverages large models to optimize RTL code. A circuit partition pipeline is utilized for fast synthesis and efficient rewriting. A multi-modal program analysis is proposed to incorporate vital visual diagram information as optimization cues. A specialized search engine is designed to identify useful optimization guides, algorithms, and code snippets that enhance the model's ability to generate optimized RTL. Additionally, we introduce a Cost-aware Monte Carlo Tree Search (C-MCTS) algorithm for efficient rewriting, managing diverse retrieved contents and steering the rewriting results. Furthermore, a fast verification pipeline is proposed to reduce verification cost. To cater to the needs of both industry and academia, we propose two benchmarking suites: the long Rewriter benchmark, targeting complex scenarios with extensive circuit partitioning, optimization trade-offs, and verification challenges, and the short Rewriter benchmark, designed for a wider range of scenarios and patterns. Our comparative analysis with established compilers such as Yosys and E-graph demonstrates significant improvements, highlighting the benefits of integrating large models into the early stages of circuit design. We provide our benchmarks at https://github.com/yaoxufeng/RTLRewriter-Bench.

1 INTRODUCTION

Optimizing Register Transfer Level (RTL) code is an essential step in the early stages of circuit design. This process involves multiple rounds of rewriting original RTL code snippets into optimized versions based on optimization patterns or synthesis feedback. Conventionally, this process relies heavily on the expertise of seasoned engineers. However, the growing complexity of design patterns has significantly hindered the efficiency of manual optimization. In comparison, existing compiler-based methods exhibit limited scope and effectiveness in optimizing complex designs, and fall short in optimizing code via synthesis feedback. Figure 1 illustrates a classic example of MUX optimization, where the revised version achieves a reduction in area by eliminating an adder. Nonetheless, certain open-source compilers, such as Yosys [1], struggle to effectively manage such scenarios.

Previous works on RTL code optimization mainly focus on specific scenarios such as data-path, MUX, memory [2–6]. Nevertheless, the real circuit design is more complicated, which contains many optimization trade-off that can not be easily addressed by compilers. In contrast, skilled engineers are good at rewriting the RTL code via synthesis feedback, taking manual as reference and directing the optimization process efficiently based on their experiences.

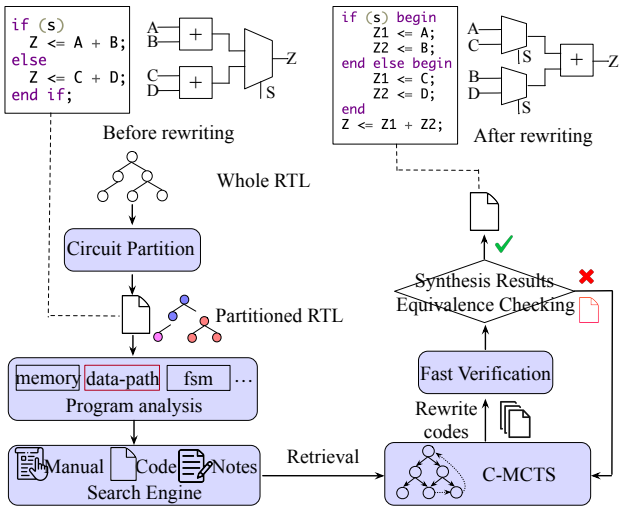

Figure 1: Rewriting example and Rewriter Pipeline.

To bridge this gap, we introduce RTLRewriter, a framework that leveraging large models for **whole RTL Code optimization process**. Figure 1 illustrates proposed RTLRewriter pipeline. We have developed multiple components within the framework to address various challenges.

- **Circuit Partition:** Large models often struggle with long contexts, which is a significant challenge in RTL code optimization. To mitigate this, we implement a circuit partitioning pipeline that breaks down the entire circuit into smaller, manageable segments, enabling faster synthesis and more effective rewriting.
- **Semantic Extraction:** Current compiler-based methods fall short in extracting meaningful semantic information from the code, particularly in terms of visual information. We address this limitation by proposing a multi-modal program analysis technique that enhances semantic understanding.
- **Documentation Utilization:** Traditional compilers generally underutilize extensive documentation, notes, and historical code. We propose a dedicated search engine to retrieve relevant content that assists in large model generation.
- **Cost-Effective Rewriting:** Not all retrieved content is beneficial for model generation, and optimal rewriting often requires multiple iterations with suitable prompts. Given the high inference costs associated with large models, we introduce a

Xufeng Yao and Yiwen Wang are equally contributed.

This work is licensed under a Creative Commons Attribution International 4.0 License.

ICCAD '24, October 27–31, 2024, New York, NY, USA
© 2024 Copyright is held by the owner/author(s).
ACM ISBN 979-8-4007-1077-3/24/10.
https://doi.org/10.1145/3676536.3676775

Cost-aware Monte Carlo Tree Search (C-MCTS) algorithm to efficiently determine the best rewriting strategies.
- **Verification Cost Reduction:** To reduce verification costs, we leverage the program analysis capabilities of large models to select the most appropriate solver for verification process.

To meet the requirements of both industry and academia, we put forth two RTL rewriting benchmarks in this study. These benchmarks encompass various examples, consisting of the original RTL code and its corresponding optimized version. The long benchmark comprises several extensive RTL cases, such as CPU designs and neural networks, presenting significant challenges akin to real-world industry scenarios. In contrast, the Small benchmark encompasses multiple small RTL cases with different scenarios, each exhibiting distinct optimization patterns. Further descriptions and details regarding these benchmarks will be provided in the experiments section.

Our major contributions are summarized as follows.

- We introduce the first LLM-aided RTL code optimization framework, leveraging the capabilities of large language models for RTL code optimization.
- Our framework incorporates multiple components to tackle various challenges.
- We propose two RTL code optimization benchmarks, curated by senior Verilog engineers.
- With experiments on different benchmarks, we proved our framework can significantly improve the synthesis performance compared with other LLM-based baselines and competitive compilers such as Yosys and E-graph.

2 RELATED WORKS

2.1 RTL Code Optimization

Previous works, such as [2–6, 6–19],have made substantial efforts toward RTL optimization, thoroughly optimizing RTL code remains challenging. In real industry applications, RTL code optimization heavily relies on experienced Verilog engineers, often requiring multiple iterations of modification based on synthesis feedback. In contrast, compiler-based methods fall short in modifying code via synthesis feedback and are limited in handling complex patterns. To address this gap, we introduce a large model-aided RTL code optimization approach, aiming to provide new directions for automatic RTL code optimization.

2.2 Large Models aided design and Challenges

The utilization of large models into electronic design automation (EDA) is an emerging field, as evidenced by recent advancements in hardware design [20–23], EDA script generation [24], RTL code generation and debugging [25–29], and other applications [30–34] These developments highlight the potential of large models to enhance and accelerate circuit design processes. However, the area of RTL code optimization, characterized by complex patterns and significant verification challenges, remains underexplored. This paper introduces a novel framework that leverages large models to address these issues, effectively filling this research gap.

Additionally, the evaluation of large models in practical industry applications presents new challenges. For instance, the Verilog generation benchmark, VerilogEval [35], uses overly simplistic cases from HDLbits. To address this, we introduce two new RTL code optimization benchmarks that include a wider range of case complexities, designed to meet the needs of both academia and industry.

3 METHODOLOGIES

3.1 Circuit Partition

The challenge of understanding long contexts in large models, due to their quadratic complexity, remains unresolved [36]. This issue also arises in RTL code optimization. Meanwhile, partitioning large circuits is a standard practice to mitigate high synthesis time costs in industry. This process involves dividing a circuit into smaller, parallelizable parts. An effective partitioning strategy should optimize total synthesis time while minimizing performance loss. Initially, the RTL code is transformed into an abstract syntax tree (AST) and then into an instance tree, where each node represents a Verilog module. A circuit predictor then estimates the synthesis time for each node. The circuit partitioner uses these predictions to balance the synthesis workload, minimizing performance loss. Finally, a scheduling algorithm efficiently allocates synthesis time slots to sub-circuits, ensuring optimal resource utilization and minimizing overall synthesis time.

Circuit Predictor. To evaluate the synthesis time of each module, we propose to establish a circuit predictor to predict the related synthesis time of each node. For RTL code, arithmetic and control operations such as binary and unary operators, conditionals, and element-selects with varying bit-widths (e.g., Add, Equality, ArithmeticShiftLeft, ElementSelect) are heavily utilized for word-level and bit-level synthesis. After building and simplifying the Abstract Syntax Tree (ast), the tree is traversed, and the bit-widths of ast nodes are compiled into a high-dimensional feature vector for subsequent prediction.

We train our circuit predictor in an offline manner where real industry-level data is leveraged. We adopt XGBoost [37] as the predictor and train the model to predict the node weights. Edge weight prediction involves estimating the performance, power, and area (PPA) effects using edge weights. These weights are calculated based on the type of connection between instances in a Verilog module, classified as direct, combinational, or sequential. This process supports workload prediction and chip partitioning.

Circuit Partitioner. To optimize parallel synthesis efficiency, we focus on partitioning the instance tree to balance synthesis time and PPA, which can be formulated as:

$$\begin{aligned} \min_{S} \quad & C = L + \lambda E \\ s.t. \quad & N_{min} \leq |S| \leq N_{max}, \end{aligned} \tag{1}$$

where C is the total cost, L represents overall synthesis time and E denotes the cost of edgecut. The edgecut is closely related to the granularity of the RTL code, and we generally prefer smaller granularity. S is the set of edgecuts, where the cardinality of S is equal to number of partitions. N_{min} and N_{max} represent the max and min number of partitions. By solving Equation (1), we aim to achieve balanced partitions with minimal synthesis time and optimal edgecut.

To address the problem, we propose a hierarchical tree partitioning algorithm. It starts by partitioning the instance tree in a top-down manner, where we partition the root node and associated sub-trees. Then, we evaluate the cost C using a bin-packing algorithm with a first-fit strategy. Subsequently, We iteratively partition the sub-tree with the largest weight and evaluate the cost C. The partitioning process continues until it meets the maximum partition number or the cost does not improve further. After partition, we transform ast to Verilog Code for further rewriting.

3.2 Large Models aided RTL Rewriting

Program Analysis and Pattern Recognition. Program analysis serves as a crucial role in circuit design. However, current compiler-based approaches fall short in extracting semantic information from raw code. For example, Visual information is an important clue in optimizing RTL code, Figure 2 shows a motivated visual example from which it's easy to recognize there existing optimization area for reducing critical path.

Inspired by chain-of-thought (COT) [38], we propose a large multi-modal model (LMM) COT program analysis pipeline. The pipeline first prompts large models to provide an in-depth analysis of diagrams based on the comprehension of given RTL code. The insights of proposed chain of analysis stem from the observation that large models can provide more accurate analysis of diagram when equipped with related RTL code, and combing diagram and RTL code can provide more informative and targeted guidance. Defining large models as π_θ, visual diagram as x_v, RTL code as x_c, initial prompt as p_{init}, and the optimization and verification patterns as p_{opt} and p_{ver}, respectively, the process can be formulated as follows:

$$\begin{aligned} p_{chain} &= \pi_\theta(x_v, x_c, p_{init}), \\ p_{opt}, p_{ver} &= \pi_\theta(p_{chain}, x_v, x_c, p_{out}), \end{aligned} \tag{2}$$

where p_{chain} represents the chain of thoughts output, and p_{output} denotes the final output prompt.

While program analysis can be applied to various scenarios in the RTL optimization process, this research primarily focuses on utilizing program analysis for optimization and verification pattern recognition. Given an RTL code and its associated diagram, we aim to leverage program analysis to identify relevant optimization patterns, such as data-path categorization and sub-expression elimination directions. Regarding verification, the large models aim to determine the type of circuit (e.g., combinational circuit, arithmetic) to enable the use of more appropriate solvers for fast verification, as detailed in Section 3.3. The structured output is produced by the specifically designed prompt such as "please return in the following format <```Optimization pattern>your response<```>" allowing for the extraction of key elements from specific patterns.

Search Engine. Retrieval augmented generation (RAG) [39] effectively enhances the generation capabilities of large models. To facilitate this, we establish a RTL database that stores relevant diagrams, codes, optimization instructions, and algorithms. The details of database creation require a considerable amount of manual effort and are omitted due to page limits.

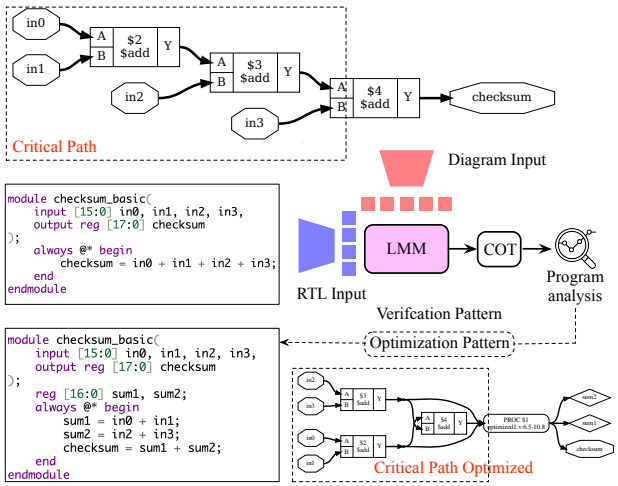

Figure 2: Multi-modal Program Analysis

Table 1: Different Retrieval types and Query methods

Type	Representation	Query Method
Diagram	Visual encoder's embedding	Join Query
Code	TF-IDF & LLM's embedding	Direct Query
Optimization Instruction	LLM's embedding	Direct Query
Optimization Algorithm	Text string	Join Query

Table 1 presents the various retrieval types, their representations, and query methods employed in our approach. For diagrams, we store the embeddings generated by a large vision model. As visual inputs are not directly suitable for providing optimization instructions, we consider them as bridges connecting related optimization instructions and codes. Consequently, we employ a join query approach [40], where the visual input is first searched, and then its mapped optimization instructions and codes are retrieved. For codes and optimization instructions, we primarily utilize the embeddings generated by language models. Additionally, we incorporate the traditional TF-IDF [41] representation for code search to enhance retrieval effectiveness. When it comes to algorithms, we adopt a join query method. In this approach, optimization instructions or codes are initially searched and then serve as a bridge to retrieve related algorithms.

C-MCTS. Although search engines can retrieve a wealth of contents, including code, optimization instructions, and algorithms, we have observed that supplying large models with all the search results as context does not consistently yield satisfactory outcomes, and can sometimes even degrade performance.

Nevertheless, selecting the suitable retrieval content is not easy and sometimes the good generation results depend on combination of several retrieval contents. Additionally, an optimal rewriting instance often requires multiple iterations, which causes high inference cost of large models. In our case, the initial selection of RAG contents involves 7 states, the rewriting codes are generated by several times (e.g., 10) due to the randomness of large models generation and the final rewriting code needs to be generated by multiple rounds in average. Consequently, the total number of states

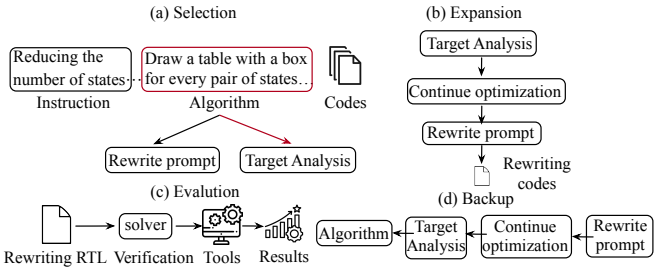

Figure 3: C-MCTS Pipeline

is over 1000 and it's impractical to utilize large models in a brute-force manner. To address the problem, we propose a Cost-aware Monte Carlo Tree Search (C-MCTS) algorithm for proficient and efficient exploration while enabling strategic selection of retrieval contents and rewriting prompts. The key elements of C-MCTS in our framework are illustrated as follows:

- **State** s: represents current status of the problem. The initial state s_0 represents the initial selection of retrieval content combined with designed prompt and RTL code, while the intermediate states represent the all sending and generating contents in this state.
- **Action** a: represents the designed prompt that large models used for generation. We design four prompt templates in action as illustrated in Table 2.
- **Value function** $Q(s, a)$: denotes values of state-action pairs (s, a) based on the synthesis results of rewriting RTL code.
- **Cost function** $C(s, a)$: estimates large models inference cost in state-pair (s, a).

As shown in Figure 3, the C-MCTS algorithm encompasses four key phases: selection, expansion, evaluation and propagation. We provide details of each process below:

Selection.: In the selection phase, the algorithm begins at the root state and proceeds to choose an action a^* from designed action set $\mathcal{A}$ as shown in Table 2. The initial root state contains 7 branches involving all combinations of retrieved content. To encourage lower-cost inference, we assign a higher initial value to the states with less retrieved content. For instance, the state value is set to 1 for retrieved code instances only, $\frac{1}{2}$ for the combination of optimization instructions and code instances, and $\frac{1}{3}$ for the combination of optimization instructions, code instances, and algorithms. The selection aims to choose the most appropriate state to continue rewriting with minimal cost estimation, as follows:

$$a^* = \underset{a}{\operatorname{argmax}}(Q(s, a) + \lambda U(s, a) + \gamma C(s, a)), \tag{3}$$

where $Q(s, a)$ denotes the expected reward for the current state, $U(s, a)$ quantifies the associated uncertainty [42], $C(s, a)$ represents the estimated cost function.

For each state, the value is based on its rewriting results. if the rewriting code can pass the verification and obtain better synthesis results then original code, the value is 1, otherwise the reward is 0.5. The value is set to 0 if the rewriting code does not pass the verification, given by:

$$Q(s, a) = \begin{cases} 0 & \text{verification failed,} \\ 0.5 & \text{equivalent results lower than original RTL,} \\ 1 & \text{equivalent results surpass original RTL,} \end{cases} \tag{4}$$

The value of state s is then updated by summation of previous values and then divided by number of selections of this state.

Uncertainty function U is given by:

$$U(s, a) = \sqrt{\frac{2\log(T)}{N(s, a)}}, \tag{5}$$

where T is total selection times, $N(s, a)$ is the number of selections of this state branch. Equation (5) is the classical UCB algorithm that can be derived by Hoeffding's inequality [42]. As $N(s, a)$ increases, the uncertainty of related state decreases. Uncertainty measures the exploration degree, it the associated factor λ is larger, we encourage more exploration and vice versa.

For the cost function, we consider both the historical cost and the estimated cost for successful rewriting in the future, defined as follows:

$$\begin{aligned} C(s, a) &= v(s, a) \cdot (1 + w(s, a)), \quad \text{and} \\ w(s, a) &= \frac{1}{1 + e^{-10 \cdot (Q(s,a) - k)}}. \end{aligned} \tag{6}$$

where $v(s, a)$ denotes the success rate of state s. Denote the number of rewriting results that can pass the verification as n, then the success rate is calculated by $\frac{n}{N(s,a)}$. $w(s, a)$ measures the estimated cost to successfully rewrite the code. $w(s, a)$ is designed like a reinforced sigmoid function and k is the balance score factor. If the value of this state $Q(s, a)$ exceeds the balance score factor k, we anticipate a lower cost for successful rewriting, and thus lower the estimated cost accordingly. Conversely, if the value of the state $Q(s)$ is less than k, the estimated cost is increased.

Expansion and Evaluation.: After the selection phase, we expand to the state s for further exploration. Instead of training a policy network like in traditional reinforcement learning, we leverage large models to decide the next action based on pre-designed options, as shown in Table 2. The action selection depends on the structured output generated by the large models for state s using a specially designed prompt, similar to those used in program analysis. The expansion continues until a rewriting action is chosen. After rewriting, we use synthesis tools to evaluate performance and check equivalence between the original and rewritten code.

Backpropagation.: After obtaining verification and synthesis results, we update the value of state s as illustrated in Equation (4).

We iteratively adopt each process and append the rewriting codes that can pass verification and surpass original Code's synthesis results in a list. When there's no obvious improvement or meeting

Table 2: Actions Description

Action	Description
Rewrite prompt	Directly rewriting RTL code.
Optimization analysis	Analyze optimization target, e.g., state reduction.
Continue optimization	Prompt large models to continue optimizing targets.
Reflection	Carefully self-checking the rewriting codes.

defined selection times, we stop the search process and choose the best rewriting instances.

3.3 Fast Verification

Verification is a critical aspect of circuit design, ensuring the correctness of implemented functionality. In our framework, verification becomes even more essential due to the inherent randomness generated by large models. A primary focus in RTL optimization frameworks is equivalence checking [43], which validates the functional equivalence between the original Verilog code and its rewritten version. Modern open-source compilers primarily use SAT-based approaches [44], translating the equivalence problem into a SAT problem. If the SAT solver finds the problem unsatisfiable, the RTL codes are not equivalent. However, SAT's worst-case computational complexity is NP-complete, meaning the time required to solve a problem can grow exponentially with its size. For example, some arithmetic circuits can lead to unexpectedly long verification times due to the complexity of their Boolean representations. In contrast, such arithmetic circuits can be efficiently verified using symbolic algebraic methods [45], which leverage algebraic techniques to simplify and compare these expressions.

Due to the randomness of large model-generated rewritten codes, conducting equivalence checking for all cases is costly. To address this, we leverage fuzz testing [46], which generates random test cases to simulate Verilog codes. By comparing the outputs of the original and rewritten code, we can identify differences and filter out problematic cases, reducing the need for extensive verification. This approach helps optimize the RTL optimization framework by avoiding the burden of exhaustive verification. After generating multiple rewritten codes via large models, we employ open-source tools to randomly generate test cases. We then filter out those rewritten cases whose outputs differ from the original. Subsequently, we utilize the verification pattern generated by program analysis as detailed in Figure 2 to determine the Verilog code is combinational or sequential and choose the appropriate solver.

4 EXPERIMENTS

4.1 Experimental Settings

Benchmarks. In this study, we introduce two benchmarks for RTL code optimization, designed to address both long and short code scenarios, crafted by experienced Verilog engineers. These benchmarks derive their optimization patterns from a comprehensive review of internal industry documents and a survey of approximately 50 scholarly articles on RTL optimization. The benchmarks encompass real-world scenarios, refined to illustrate typical industry challenges, with optimized solutions provided by skilled Verilog engineers. For the short benchmark suite, out of an initial 55 cases, 14 representative scenarios were selected for detailed analysis in our experimental results due to page constraints. These scenarios cover various RTL code aspects, including basic patterns, data-path, memory, MUX, FSM, and control logic. The long benchmarks include five extensive cases involving CPU design, neural networks, and image processing, which is more challengeable.

Baselines. We compare RTLRewriter with 6 baselines in three types approaches as follows:

- **Two SOTA large models**: We evaluate RTLRewriter in comparison to two leading large-scale models including GPT4 [47] and Claude3-Opus. We access GPT-4 and Claude3-Opus via Poe (https://poe.com).
- **Two SOTA open-source large RTL models**: RTLRewriter is also compared with two specialized language models designed for RTL code generation: VeriGen [25] and RTLCoder [26] are two currently sota large language models targeting RTL code generation.
- **Two competitive compilers**: Additionally, we assess RTLRewriter against two widely-used open-source compilers: Egg [48] and Yosys [1].

To ensure fair and consistent evaluation, all language model baselines are tested using **identical prompt engineering techniques**. For the compilers, we utilize the standard optimization commands in Yosys. We implement our own version of Egg integrated with Yosys, incorporating custom optimization optimization operators, which serves as a stronger baseline than Egg alone.

Implementations. In this research, we introduce the RTLRewriter framework, which leverages GPT-4V [47], a state-of-the-art multi-modal and language model, for multi-modal program analysis and RTL code optimization. Our search engine components include ViT [49] as the image encoder, LLama3 [50] for text embeddings, and DeepSeek [51] for code embeddings. For verification, we employ iVerilog [52] to compile the code, alongside self-developed scripts for generating test cases. Additionally, we utilize ABC [53] and egg as our solvers, where ABC excels in SAT-based verification, while egg is highly efficient for arithmetic circuit verification.

Evaluation Metrics. For the overall optimization system, we focus on synthesis results and whole run-time. We also take other metrics that measures the effectiveness of each part. The calculations of these metrics are listed below: **Wires**: represent the interconnections between components, with a higher count indicating more complex routing. **Cells**: represent the logical components used in the design, with a higher count indicating more logic complexity. **Area**: total hardware resources utilized by the synthesized circuit. This includes the number of logic gates, flip-flops, and interconnections required to implement the design, generated by ABC [53]. **Delay**: measures the longest propagation time of signals through a circuit from input to output, generated by ABC [53].

4.2 Performance Analysis

Table 3 illustrates the results of short RTL rewriting benchmarks across 14 cases, encompassing both baseline results and human implementations, validated by senior Verilog engineers. Wires and cells are two critical evaluation metrics in front-end RTL optimization due to their strong correlation with area and delay. In real-world applications, optimizing these metrics, even by 1%, represents significant progress. We adopt Yosys results as our baseline, where fewer wires and cells indicate better results. The results with gray color mean that the method fails to optimize the code. Notably, our method outperforms all competitive baselines by a substantial margin. Among these cases, state-of-the-art large models like GPT4 and Claude3-Opus exhibit great potential in optimizing RTL code, even surpassing curated compiler-based solutions such as Yosys+Egg.

Table 3: Comparisons of baseline approaches on short RTL rewriting benchmarks

Test Case ID	Yosys		Yosys+Egg		GPT4		Claude3		VeriGen		RTLCoder		RTLRwriter	
	Wires	Cells	Wires	Cells	Wires	Cells	Wires	Cells	Wires	Cells	Wires	Cells	Wires	Cells
case1	28	18	24	14	28	18	28	18	28	18	28	18	24	14
case2	11646	11824	11307	11478	11507	11684	11609	11787	11684	11824	11588	11766	11299	11477
case3	1136	1220	1095	1176	890	974	910	992	1135	1120	1136	1120	890	974
case4	1376	1462	1327	1409	1376	1462	1376	1462	1376	1462	1376	1462	1127	1213
case5	193	49	164	49	193	49	193	49	193	49	193	49	65	49
case6	172	129	161	129	171	129	171	129	172	129	172	129	161	129
case7	402	403	378	379	402	403	386	387	402	403	402	403	353	354
case8	466	354	434	330	466	354	466	354	466	354	466	354	370	354
case9	70	71	67	68	70	71	70	71	70	71	70	71	34	32
case10	59	56	56	54	59	56	59	56	59	56	59	56	41	42
case11	34	35	33	34	21	24	34	35	34	35	34	35	21	24
case12	14782	14960	14782	14960	14782	14960	14695	14873	14782	14960	14782	14960	14525	14703
case13	7	2	7	2	3	1	3	1	7	2	7	2	3	1
case14	16	6	14	5	8	3	8	3	16	6	16	6	8	3
GeoMean	222.68	161.97	209.14	153.22	189.19	140.43	195.56	144.04	222.68	161.97	222.60	161.91	152.83	124.46
Ratio	1.00	1.00	0.93	0.94	0.85	0.87	0.88	0.89	1.00	1.00	0.99	0.99	0.69	0.77

Table 4: Comparisons of baseline approaches on long RTL rewriting benchmarks

Test Case ID	Yosys		Yosys+Egg		GPT4		Claude3		VeriGen		RTLCoder		RTLRwriter	
	Area	Delay	Area	Delay	Area	Delay	Area	Delay	Area	Delay	Area	Delay	Area	Delay
CPU	179025.72	1989.76	167996.88	1688.54	179025.72	1989.76	179025.72	1989.76	179025.72	1989.76	179025.72	1989.76	167634.27	1592.58
CNN	26071.46	15890.42	22004.64	14746.34	20104.01	15890.42	20104.01	15890.42	20104.01	15890.42	20104.01	15890.42	20104.01	13565.95
FFT	71385.35	184098.72	60321.54	183545.68	71385.35	184098.72	71385.35	184098.72	71385.35	184098.72	71385.35	184098.72	56451.58	181495.83
Huffman	106045.69	1544.00	97480.22	1544.36	106045.69	1544.00	106045.69	1544.00	106045.69	1544.00	106045.69	1544.00	99142.98	1545.64
VMachine	1212.43	569.20	1030.23	642.87	1212.43	569.20	1212.43	569.20	1212.43	569.20	1212.43	569.20	799.60	676.81
GeoMean	33602.22	5517.98	29513.98	5387.19	33602.22	5517.98	33602.22	5517.98	33602.22	5517.98	33602.22	5517.98	27270.39	5279.57
Ratio	1.00	1.00	0.87	0.97	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.81	0.96

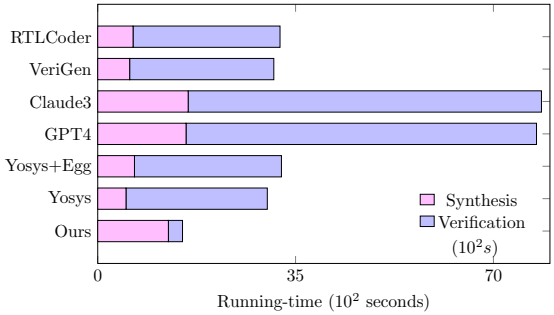

Figure 4: Running-time Comparison

However, without the aid of additional techniques, these raw large models struggle to optimize diverse cases. Additionally, we have observed that the large RTL models targeting code generation such as [26] and VeriGen [25] can hardly optimize the RTL code even they are fine-tuned from large corpus of RTL dataset. In contrast, our method achieves optimization in nearly all cases. Interestingly, in some cases, such as case 14, the large models generate a superior implementation compared to the human version, underscoring the potential for leveraging large models in RTL code optimization.

Table 4 presents the results for the long-benchmark. These cases are designed to simulate real industry applications, featuring significantly longer code lines. We observe that the proposed RTLRewriter continues to outperform all baseline methods by a substantial margin. Due to the complexity of the designs, GPT-4 and Claude 3-Opus were less effective in optimizing RTL code. However, the Yosys+Egg approach consistently demonstrated improvements. Unlike the short cases, the long cases are more complex, resulting in smaller overall improvements.

Another important evaluation metric in RTL optimization scenarios is running time, which encompasses both synthesis time and verification time. As shown in Figure 4, our method achieves significantly faster overall running time. Compared with other approaches based on large models such as GPT4 and Claud3-Opus, our method achieves much faster synthesis time, thanks to our fuzz testing strategy. Compared to other compiler-based methods, our approach significantly reduces verification time, especially for arithmetic circuits where the large UNSAT core can consume considerable time. Large RTL models like VeriGen and RTLCoder achieve similar synthesis and verification runtimes as compiler-based methods, largely because they often generate repetitive original RTL code. Despite this, our method consistently outperforms these approaches.

5 CONCLUSION

In this paper, we present an automatic RTL code optimization framework leveraging large models. We introduce several key components to address the challenges, including a circuit partition pipeline, large model-aided RTL optimization encompassing multi-modal program analysis, a search engine, and a cost-aware search algorithm for efficient rewriting. Additionally, we have developed a fast verification pipeline to streamline the verification process and reduce costs. Moreover, we have created two datasets to foster further advancements in RTL code optimization. We hope our work can stimulate innovation in RTL code optimization.

REFERENCES

[1] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, vol. 97, 2013.

[2] R. Pasko, P. Schaumont, V. Derudder, S. Vernalde, and D. Durackova, "A new algorithm for elimination of common subexpressions," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 1, pp. 58–68, 1999.

[3] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 2, pp. 181–210, 1991.

[4] D. Chen and J. Cong, "Register binding and port assignment for multiplexer optimization," in *ASP-DAC 2004: Asia and South Pacific Design Automation Conference 2004 (IEEE Cat. No. 04EX753)*. IEEE, 2004, pp. 68–73.

[5] P. Pišteka, K. Jelemenská, and M. Kolesár, "Reduction of multiplexer trees using modified lookup table."

[6] Z. Wang, H. You, J. Wang, M. Liu, Y. Su, and Y. Zhang, "Optimization of multiplexer combination in rtl logic synthesis," in *2023 International Symposium of Electronics Design Automation (ISEDA)*. IEEE, 2023, pp. 121–125.

[7] J. Cocke, "Global common subexpression elimination," in *Proceedings of a symposium on Compiler optimization*, 1970, pp. 20–24.

[8] J. Knoop, O. Rüthing, and B. Steffen, "Partial dead code elimination," *ACM Sigplan Notices*, vol. 29, no. 6, pp. 147–158, 1994.

[9] K. D. Cooper, L. T. Simpson, and C. A. Vick, "Operator strength reduction," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 23, no. 5, pp. 603–625, 2001.

[10] A. Darte, "On the complexity of loop fusion," in *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00425)*. IEEE, 1999, pp. 149–157.

[11] J. Teifel and R. Manohar, "Highly pipelined asynchronous fpgas," in *proceedings of the 2004 ACM/SIGDA 12th International symposium on field programmable gate arrays*, 2004, pp. 133–142.

[12] T. Kam, T. Villa, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, *Synthesis of finite state machines: functional optimization*. Springer Science & Business Media, 2013.

[13] T. Villa, T. Kam, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, *Synthesis of finite state machines: logic optimization*. Springer Science & Business Media, 2012.

[14] R. S. Shelar, M. P. Desai, and H. Narayanan, "Decomposition of finite state machines for area, delay minimization," in *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No. 99CB37040)*. IEEE, 1999, pp. 620–625.

[15] C. E. LaForest and J. G. Steffan, "Efficient multi-ported memories for fpgas," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, 2010, pp. 41–50.

[16] J. Ma, G. Zuo, K. Loughlin, X. Cheng, Y. Liu, A. M. Eneyew, Z. Qi, and B. Kasikci, "A hypervisor for shared-memory fpga platforms," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 827–844.

[17] Y. Zhou, K. M. Al-Hawaj, and Z. Zhang, "A new approach to automatic memory banking using trace-based address mining," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 179–188.

[18] B.-C. Lai, B.-Y. Chen, B.-E. Chen, and Y.-D. Hsin, "Remap+: An efficient banking architecture for multiple writes of algorithmic memory," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 3, pp. 660–671, 2019.

[19] J. Park and P. C. Diniz, "Synthesis of pipelined memory access controllers for streamed data applications on fpga-based computing engines," in *Proceedings of the 14th international symposium on Systems synthesis*, 2001, pp. 221–226.

[20] M. Liu, T.-D. Ene, R. Kirby, C. Cheng, N. Pinckney, R. Liang, J. Alben, H. Anand, S. Banerjee, I. Bayraktaroglu *et al.*, "Chipnemo: Domain-adapted llms for chip design," *arXiv preprint arXiv:2311.00176*, 2023.

[21] J. Blocklove, S. Garg, R. Karri, and H. Pearce, "Chip-chat: Challenges and opportunities in conversational hardware design," in *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*. IEEE, 2023, pp. 1–6.

[22] Y. Fu, Y. Zhang, Z. Yu, S. Li, Z. Ye, C. Li, C. Wan, and Y. C. Lin, "Gpt4aigchip: Towards next-generation ai accelerator design automation via large language models," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.

[23] Y. Zhang, Y. Fu, Z. Yu, K. Zhao, C. Wan, C. Li, and Y. C. Lin, "Data4aigchip: An automated data generation and validation flow for llm-assisted hardware design," 2024.

[24] H. Wu, Z. He, X. Zhang, X. Yao, S. Zheng, H. Zheng, and B. Yu, "Chateda: A large language model powered autonomous agent for eda," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[25] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, "Verigen: A large language model for verilog code generation," *arXiv preprint arXiv:2308.00708*, 2023.

[26] S. Liu, W. Fang, Y. Lu, Q. Zhang, H. Zhang, and Z. Xie, "Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution," *arXiv preprint arXiv:2312.08617*, 2023.

[27] M. DeLorenzo, A. B. Chowdhury, V. Gohil, S. Thakur, R. Karri, S. Garg, and J. Rajendran, "Make every move count: Llm-based high-quality rtl code generation using mcts," *arXiv preprint arXiv:2402.03289*, 2024.

[28] Y. Tsai, M. Liu, and H. Ren, "Rtlfixer: Automatically fixing rtl syntax errors with large language models," *arXiv preprint arXiv:2311.16543*, 2023.

[29] Z. Pei, H.-L. Zhen, M. Yuan, Y. Huang, and B. Yu, "Betterv: Controlled verilog generation with discriminative guidance," *arXiv preprint arXiv:2402.03375*, 2024.

[30] Z. Wang, L. Chen, J. Wang, X. Li, Y. Bai, X. Li, M. Yuan, J. Hao, Y. Zhang, and F. Wu, "A circuit domain generalization framework for efficient logic synthesis in chip design," *arXiv preprint arXiv:2309.03208*, 2023.

[31] Z. Wang, J. Wang, D. Zuo, J. Yunjie, X. Xia, Y. Ma, H. Jianye, M. Yuan, Y. Zhang, and F. Wu, "A hierarchical adaptive multi-task reinforcement learning framework for multiplier circuit design," in *Forty-first International Conference on Machine Learning*, 2024.

[32] T. Liu, Y. Sun, L. Chen, X. Li, M. Yuan, and E. F. Young, "A unified parallel framework for lut mapping and logic optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[33] X. Li, X. Li, L. Chen, X. Zhang, M. Yuan, and J. Wang, "Logic synthesis with generative deep neural networks," *arXiv preprint arXiv:2406.04699*, 2024.

[34] ——, "Circuit transformer: End-to-end circuit design by predicting the next gate," *arXiv preprint arXiv:2403.13838*, 2024.

[35] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "Verilogeval: Evaluating large language models for verilog code generation," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–8.

[36] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 2024.

[37] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," 2016, pp. 785–794.

[38] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.

[39] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, and H. Wang, "Retrieval-augmented generation for large language models: A survey," *arXiv preprint arXiv:2312.10997*, 2023.

[40] Y. E. Ioannidis, "Query optimization," *ACM Computing Surveys (CSUR)*, vol. 28, no. 1, pp. 121–123, 1996.

[41] A. Aizawa, "An information-theoretic perspective of tf–idf measures," *Information Processing & Management*, vol. 39, no. 1, pp. 45–65, 2003.

[42] P. Auer, "Using confidence bounds for exploitation-exploration trade-offs," *Journal of Machine Learning Research*, vol. 3, no. Nov, pp. 397–422, 2002.

[43] L. Lavagno, I. L. Markov, G. Martin, and L. K. Scheffer, *Electronic design automation for IC implementation, circuit design, and process technology: circuit design, and process technology*. CRC Press, 2016.

[44] J. Marques-Silva and T. Glass, "Combinational equivalence checking using satisfiability and recursive learning," in *Proceedings of the conference on Design, automation and test in Europe*, 1999, pp. 33–es.

[45] A. Sayed-Ahmed, D. Große, M. Soeken, and R. Drechsler, "Equivalence checking using gröbner bases," in *2016 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2016, pp. 169–176.

[46] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[47] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, Aleman *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[48] S. Coward, G. A. Constantinides, and T. Drane, "Automatic datapath optimization using e-graphs," in *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*. IEEE, 2022, pp. 43–50.

[49] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[50] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[51] X. Bi, D. Chen, G. Chen, S. Chen, D. Dai *et al.*, "Deepseek llm: Scaling open-source language models with longtermism," *arXiv preprint arXiv:2401.02954*, 2024.

[52] S. Williams and M. Baxter, "Icarus verilog: open-source verilog more than a year later," *Linux Journal*, vol. 2002, no. 99, p. 3, 2002.

[53] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*. Springer, 2010, pp. 24–40.