CombRewriter: Enabling Combinational Logic Simplification in MLIR-Based Hardware Compiler

Haisheng Zheng [1] Zhuolun He [2,3] Shuo Yin [2] Yuzhe Ma [1] Bei Yu [2]
[1] HKUST (GZ) [2] CUHK [3] ChatEDA Tech

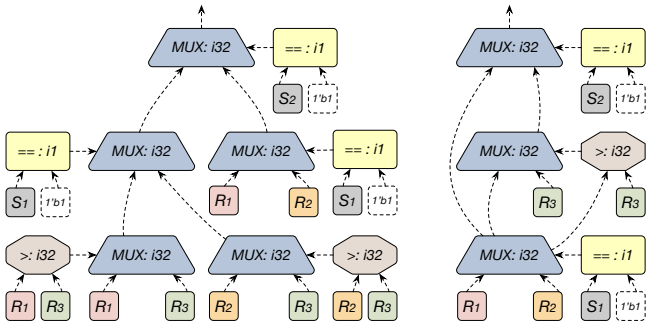
***Abstract*—Modern Hardware Description Languages (HDLs) play a pivotal role in enabling swift and adaptable hardware development. A hardware compiler translates high-level designer intents into a concrete hardware implementation, the quality of which directly determines ultimate circuit performance. However, current hardware compilers may overlook opportunities for combinational logic simplification, leading to RTL code that contains redundant logic and degrades the Quality of Results (QoR) of the synthesized netlist. This paper presents CombRewriter, a novel approach that incorporates compilation-level optimization techniques into combinational logic simplification. Experimental results demonstrate that the proposed method effectively reduces netlist area.**

I. Introduction

The rapid progress of Very Large-Scale Integration (VLSI) technology has led to increasingly complex integrated circuits, resulting in longer design and development cycles. Traditional Hardware Description Languages (HDLs) such as VHDL, Verilog, and SystemVerilog have served as the gold standard in the hardware design stack due to their strong synthesis and verification support. However, they offer limited flexibility and productivity. As a result, there is growing demand for advanced hardware development tools that can simplify the design process and enhance efficiency. To meet these requirements and make hardware development more accessible, various hardware description languages have been introduced.

Modern HDLs such as Chisel [1], SpinalHDL [2], and MyHDL [3] have garnered significant attention for their use of software programming language constructs, streamlining Field Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs) design. Domain-specific HDLs such as Spatial [4] and Dahlia [5] specialize in accelerator design, employing advanced compilation techniques for automatic pipeline and parallel processing unit creation. These HDLs contribute to more efficient hardware designs for specific tasks. The compilers associated with these hardware description languages (HDLs) are essential for translating high-level designs into synthesis-ready Register-Transfer Level (RTL) code and ensuring that the synthesized hardware faithfully reflects the concepts envisioned by the designers.

Hardware Intermediate Representations (IRs) [5]–[15] form the foundation of these compilers. Various optimizations and transformations are performed using these IRs to guarantee high-performance RTL generation. In the case of Chisel, which is widely used in hardware design, the Circuit IR Compiler and Tools (CIRCT) [16] replaced the Scala-FIRRTL Compiler (SFC) [7] with the release of Chisel 3.6.0, improving the quality and readability of the generated Verilog. Built on the Multi-Level Intermediate Representation (MLIR) [17] framework, CIRCT eliminates the need to develop compiler infrastructure from scratch and addresses common challenges independently faced by many language projects, such as redundant code elimination.



(a) Original. (b) Transformed.

Fig. 1 An Example of Hardware Logic Simplification Showing Reduction in Multiplexers and Greater-Than Comparator.

However, CIRCT may generate RTL code that contains superfluous logic, which negatively impacts the quality of the result of netlist. Fig. 1(a) illustrates this issue with a simplified example derived from Gemmini [18], compiled using CIRCT, where the dataflow can be simplified. After simplification, as shown in Fig. 1(b), the number of multiplexers (`MUXs`) is reduced from *five* to *three*, and the number of greater-than comparators is reduced from *two* to *one*.

In the above example, the initial stage of the logic simplification process parallels the *Simplify Control Flow Graph (SimplifyCFG)* [19] process used in modern compiler optimization, whereas the subsequent stage can be implemented using the *Common Subexpression Elimination (CSE)* [20] technique within modern compilers.

Nevertheless, several factors preclude the direct application of the *SimplifyCFG* process to hardware logic simplification. One key factor is that transformations should be specifically tailored for hardware IR to ensure effective logic simplification. Moreover, performance analysis should be specifically designed for hardware IR simplification. This is further discussed in Section IV-D.

Recognizing the similarity between logic simplification in hardware IR and the *SimplifyCFG* process in modern compiler design, this paper introduces CombRewriter, a hardware IR

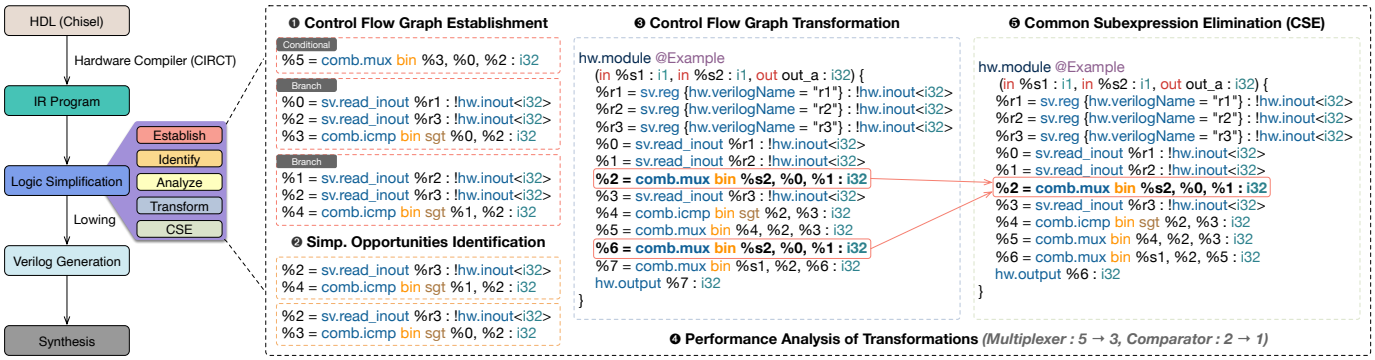


Fig. 2 The Overall Flow of CombRewriter.

simplification approach that can lead to a reduction in netlist area. The main contributions of this work are as follows:

- An approach for addressing *logic simplification* from a compiler design perspective and integrating it into an open-source compilation framework is proposed.
- Domain-specific transformations are designed to exploit hardware IR-specific simplification opportunities.
- A machine learning-assisted performance analysis approach, specifically tailored for hardware logic simplification, is introduced.
- Comprehensive experiments are conducted to demonstrate the effectiveness and efficiency of the proposed approach.

II. Preliminaries

Multi-Level Intermediate Representation (MLIR) is a compiler infrastructure that allows programs to be expressed and optimized at multiple levels of abstraction. This design enables different aspects of a system to be handled independently.

Dialect in MLIR is a customizable IR framework that enables the definition of custom elements, such as operations, types, and attributes. This flexibility enables optimizations to be applied at the appropriate level and enhances compilation efficiency. For example, since optimization techniques for combinational and sequential logic differ, CIRCT defines *comb dialects* for combinational logic and *seq dialect* for sequential logic.

Pass in MLIR is a component that performs a specific transformation or analysis on the intermediate representation, enabling systematic code optimization and lowering across different abstraction levels.

Control Flow Graph (CFG) is a fundamental data structure in compilers that represents the control flow of a program. It models the branching structure through directed edges that indicate possible execution paths. In hardware IR graph, a multiplexer operation can be conceptualized as a CFG, where the selector signal creates conditional branching behavior equivalent to an if-else statement by determining which input path to follow.

Common Subexpression Elimination (CSE) is a compiler optimization technique that identifies instances of identical expressions in a program and eliminates redundant computations. An *expression* refers to a sequence of operations that computes a value. When an expression is repeated with identical inputs, computing it multiple times is redundant.

III. Problem Formulation

A hardware design in a hardware compiler can be represented as a hardware IR graph, denoted as $G = (V, E)$, where V represents the set of vertices and E represents the set of edges. Each vertex $v \in V$ symbolizes an operator (e.g., a multiplexer, an adder) within the hardware IR graph. Each edge $(u, v) \in E$ describes the dependency between nodes u and v. Based on the hardware IR graph, the problem can be formulated as:

Problem 1 (Combinational Logic Simplification)**.** *Given a hardware IR graph $G = (V, E)$, design novel methodology based on compiler techniques to identify optimizable logic in multiplexer branches to simplify G in hardware compiler.*

IV. Algorithms

Fig. 2 illustrates the overall flow of the proposed approach, CombRewriter, which details the simplification process outlined in Fig. 1. Given a hardware design, as shown in Fig. 1(a), represented as an IR program using the *comb dialect*, the simplification process of CombRewriter consists of the following stages:

❶ **Control Flow Graph Establishment**. Operations in the hardware design that influence branching and decision-making are extracted to construct a control flow graph.

❷ **Simplification Opportunities Identification**. The control flow graph is analyzed to identify simplification opportunities.

❸ **Control Flow Graph Transformation**. If simplification opportunities are found, corresponding transformations are applied to the control flow graph to simplify the hardware IR logic.

TABLE I Common Operation Hoisting Transformations.

Category	Original	Transformed
MUX Relocation	$F = S\,?\,COMOps(I_1) : COMOps(I_2)$	$F = COMOps(S\,?\,I_1 : I_2)$
XOR Introduction	$F = S\,?\,COMOps(I_1) : NOT(COMOps(I_2))$	$F = XOR(COMOps(S\,?\,I_1 : I_2), S)$
MUX Introduction	$F = S\,?\,DIFFOps_1(COMOps(I_1)) :$ $DIFFOps_2(COMOps(I_2))$	$F = S\,?\,DIFFOps_1(F_{COM}) :$ $F_{COM} = COMOps(S\,?\,I_1 : I_2)$

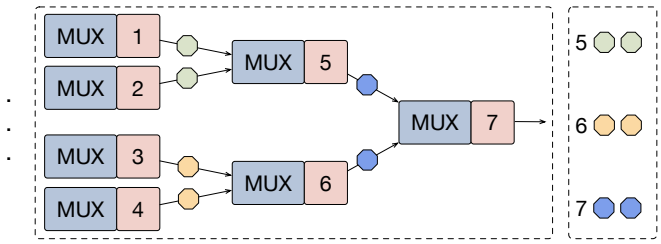
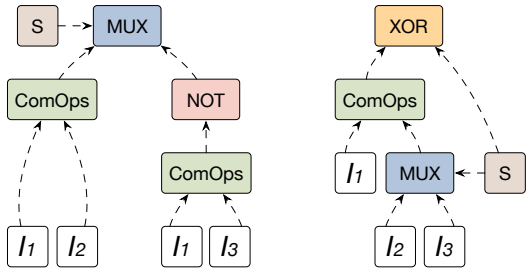


Fig. 3 Visualization of Control Flow Graphs with Simplification Order.



(a) Original. (b) Transformed.

Fig. 4 Introducing XOR Operation to Simplify CFG.

❹ **Performance Analysis of Transformations**. A machine learning-assisted performance analysis algorithm assesses the impact of potential transformations, as they may increase area or degrade timing. Transformations are applied if they provide benefits. Otherwise, the original design is retained.

❺ **Common Subexpression Elimination**. This step removes redundant operations that may be introduced during simplification, further optimizing the design.

A. Control Flow Graph Establishment

Due to the branches of multiplexers in the *comb dialect* within CIRCT not being encapsulated as *basic blocks*, it is necessary to extract the operations of these branches as *basic blocks* to establish the control flow graph for subsequent hardware IR graph simplification. Breadth-First Search (BFS) is employed to traverse the hardware IR graph and establish the control flow graph.

Once the control flow graph is established, determining the *simplification order* is crucial for ensuring the efficiency of the simplification process. The following example further illustrates this:

Example 1 (Simplification Order Matters)**.** *Consider the expression* $F = S_1?R_1 \times 2 : (S_2?R_2 \times 2 : R_3 \times 2)$. *If simplification begins from the false branch, the common operation '*$\times 2$*' is exposed and extracted, yielding* $F = (S_1?R_1 : (S_2?R_2 : R_3)) \times 2$ *in a single step. Conversely, if simplification starts from the true branch, more iterations are required to achieve the same result.*

Therefore, the dominator tree [21], widely used in modern compilers, is employed to determine the simplification order of control flow graphs. It represents dominance relationships between basic blocks in a program's control flow graph. The dominators of a basic block n can be recursively defined as:

$$\mathrm{Dom}(n) = \begin{cases} \{n\}, & \text{if } n = n_0 \\ \{n\} \cup \left(\bigcap_{p \in \mathrm{preds}(n)} \mathrm{Dom}(p)\right), & \text{if } n \neq n_0 \end{cases}. \tag{1}$$

The dominator of the initial basic block is the block itself. For any other basic block n, its dominator set contains itself as well as the intersection of the dominator sets of all its predecessors $p \in \mathrm{preds}(n)$. Fig. 3 visualizes the control flow graphs and simplification order in a hardware IR graph.

B. Optimization Opportunities Identification

The identification of simplification opportunities, as illustrated in TABLE I and TABLE II, is a pattern matching problem in compiler optimization and can be formulated as a graph isomorphism problem, where the goal is to identify equivalent structures within the given graphs. Specifically, Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are considered isomorphic if there exists a bijection $\phi : V_1 \to V_2$ such that

$$\{v, w\} \in E_1 \Leftrightarrow \{\phi(v), \phi(w)\} \in E_2. \tag{2}$$

The VF2 algorithm [22] is employed to solve this problem due to its efficiency.

Furthermore, a variant of common operation identification is designed to approximate the isomorphism by ignoring the negation (NOT) operation, which can reveal additional simplification opportunities. To further illustrate this, examine the following example.

Example 2 (Approximate Common Operation Identification by Ignoring NOT Operation)**.** *As illustrated in Fig. 4, consider the expression* $F = S?(I_1 \times I_2) : NOT(I_1 \times I_3)$. *By ignoring the NOT operation in the false branch, both branches execute a set of common operations. To hoist the common operations,*

TABLE II Domain-Specific Transformations (The $*$ Operation Represents Both $+, \times$).

Category	Original	Transformed
Arithmetic Logic Exchange	$F = S?(a + b) : c$	$F = (S?a : c) + (S?b : 0)$
	$F = S?a : (b + c)$	$F = (S?a : b) + (S?1 : c)$
	$F = (S?0 : a) \times b$	$F = a \times (S?0 : b)$
Operation Reordering	$F = ((S?c : d) * b) * a$	$F = ((S?c : d) * a) * b$
Shift Operation Distribution	$F = (S?a : c) \ll (S?b : d)$	$F = S?(a \ll b) : (c \ll d)$
	$F = (S?a : c) \gg (S?b : d)$	$F = S?(a \gg b) : (c \gg d)$

an exclusive OR (XOR) operation can be introduced. The expression is transformed into: $F = XOR(I_1 \times (S?I_2 : I_3), S)$, which removes common operations at the cost of introducing a XOR operation. When the area cost of the XOR operation is lower than that of the common operations, this transformation provides benefits.

C. Control Flow Graph Transformation

The transformed expressions are illustrated in TABLE I and TABLE II. In cases where an expression in the hardware IR graph qualifies for *Common Operation Hoisting*, CombRewriter is re-applied to ensure that the updated graph no longer matches the patterns for a *Domain-Specific Transformation*. To further illustrate this, examine the following example:

Example 3 (Operation Reordering)**.** *Consider the expression $F = S?(a \times b \times c) : (a \times b \times d)$ where $a = 16$, $b = 32$, $c = 16$, and $d = 16$ bits. Since the subexpression $a \times b$ is common to both branches, it can be hoisted to form $F = (S?c : d) \times (a \times b)$. While this transformation reduces redundancy, further optimization can be achieved through operation reordering, especially in hardware implementations where resource usage is influenced by operand bit-widths. This contrasts with software programming, where arithmetic operations are typically executed using fixed-width processing units and are therefore less sensitive to operand widths.*

Specifically, the original transformation $F = ((S?c : d) \times b) \times a$ requires two 32-bit multipliers. By reordering to $F = ((S?c : d) \times a) \times b$, the implementation requires only one 16-bit multiplier and one 32-bit multiplier, thereby reducing the area usage.

The effectiveness of this transformation depends on operand bit-widths. For instance, when all operands share the same bit width, both orderings result in equivalent resource usage and no improvement is achieved.

D. Performance Analysis of Transformation.

To ensure that transformations provide benefits, comprehensive performance analysis methods are designed to evaluate their effectiveness.

Common Operation Hoisting. In the category of *MUX Relocation*, area reduction is achieved in most cases, except when the common operations involve only *concatenation*, *extraction*, *replication*, or *shifting* with constant values. These operations can be implemented using wires without the need for logic gates, rendering the transformation unnecessary.

For the *XOR Introduction* and *MUX Introduction* categories, the transformation should not be applied if the area cost of the common operations is lower than that of the introduced XOR or MUX operations. Additionally, these transformations can increase the logic level of the circuit path, potentially degrading the timing performance of the design.

As a result, careful evaluation is required to determine whether the transformation might negatively impact timing. Specifically, if the hardware IR logic is not located on the *critical path* of the design, the transformation may be applied to simplify the hardware IR graph without affecting overall timing performance.

However, identifying whether the transformed logic lies on the critical path is challenging due to the abstraction gap between the hardware IR graph and the netlist. To address this, Section IV-E presents an approach for assessing such scenarios.

Domain-Specific Transformation In this category, not all rule-matching cases lead to improvements. The effectiveness of a transformation depends on the bit-widths of the operands.

Example 4 (Impact of Operand Bit-Width)**.** *Consider the expression $F = S?(a + b) : c$, which can be transformed into $F = (S?a : c) + (S?b : 0)$, and potentially further simplified as $F = (S?a : c) + (S \,\&\, b)$. The area usage varies significantly depending on the bit-widths of the operands.*

Case 1: *$a = 16$, $b = 64$, $c = 16$ bits. The transformation reduces the number of 2-1 MUXes from 64 to 16 and introduces 64 AND gates, resulting in better area efficiency.*

Case 2: *$a = 16$, $b = 16$, $c = 16$ bits. The transformation introduces 16 AND gates without reducing MUX usage, leading to increased area.*

Case 3: *$a = 16$, $b = 24$, $c = 16$ bits. The transformation reduces the number of 2-1 MUXes from 24 to 16 and adds 24 AND gates. Assuming one 2-1 MUX equals three AND gates in area, there is no net area benefit. However, due to variations in standard-cell libraries (i.e., a 2-1 MUX may not equal exactly three AND gates), the actual area advantage is uncertain.*

Therefore, cases with *uncertain area impact* should be further evaluated to ensure the effectiveness of the applied transformations.

TABLE III Comparison of E-morphic [23] and CombRewriter in Netlist Optimization.

Category	E-morphic [23]		CombRewriter (Ours)	
	$\Delta Area$ (%)	$\Delta Delay$ (%)	$\Delta Area$ (%)	$\Delta Delay$ (%)
MUX Reduction	-10.50%	-8.08%	-47.64%	-13.97%
XOR Introduction	-3.75%	-2.15%	-45.06%	-12.94%
MUX Introduction	-4.73%	-9.08%	-44.62%	-10.81%
Arithmetic Logic Exchange	+2.86%	+2.79%	-4.25%	-1.11%
Operation Reordering	-8.02%	-8.58%	-23.12%	-16.83%
Shift Operation Distribution	-8.71%	+26.00%	-4.64%	-8.54%

E. Machine-Learning Assisted Performance Analysis

To address the above concerns that transformations may increase design timing and result in uncertain area usage, our goal is to design an algorithm that assesses whether a transformation will gain benefits. In logic synthesis, since netlists can be represented as directed acyclic graphs (DAGs), numerous studies [24]–[28] leverage graph neural networks (GNNs) to extract meaningful features from netlists and predict their performance.

Since hardware IR graph is also represented as a DAG, we introduce SimplifyGuider, a GNN-based algorithm designed to evaluate transformation effectiveness.

Node Feature Encoding. Several hardware IR node information is used as node features, as shown below:

Operation Information. The operation type of each node along with its operand width serves as a fundamental feature.

Node Level. Node levels in the hardware IR graph play a crucial role by indicating the nodes' positions within the logical hierarchy of the circuit. This information influences both the potential for further optimization and the overall circuit structure. To capture these positional characteristics, we apply positional encoding (PE):

$$\mathrm{PE}(v) = \sin\left(\frac{level(v)}{L_{\max}}\right) + \cos\left(\frac{level(v)}{L_{\max}}\right) \tag{3}$$

where $\mathrm{PE}(v)$ denotes the scalar positional encoding value for node v, $level(v)$ refers to the logical level of node v in the circuit, and $L_{\max}$ is the maximum possible level in the graph. $L_{\max}$ is set to *10,000* in the SimplifyGuider implementation.

Hardware IR Graph Feature Extraction. GraphSAGE [29] is chosen as our GNN model to extract features from the hardware IR graph. It updates node representations by aggregating features from neighboring nodes. The core update rule of GraphSAGE is given by:

$$\boldsymbol{h_v^{(k+1)}} = \mathrm{MEAN}^{(k)}\left(\{\boldsymbol{h_u^{(k)}} \mid u \in \mathcal{N}(v)\}\right) \oplus \boldsymbol{h_v^{(k)}}, \tag{4}$$

where $\boldsymbol{h_v^{(k)}}$ represents the feature vector of node v at the k-th layer, $\mathcal{N}(v)$ denotes the set of neighboring nodes, $\mathrm{MEAN}^{(k)}$ is the aggregation function applied at layer k, and $\oplus$ indicates concatenation. This approach enables GraphSAGE to effectively capture both local and global information while updating node features efficiently.

Learning To Rank. In our context, the focus is solely on determining whether a transformation will lead to performance improvement, rather than predicting the exact performance value. Therefore, the objective is to establish a ranking among different transformation options. Ranking loss [30] is a fundamental concept in machine learning that aims to learn the relative ordering of items rather than their absolute values. It is particularly useful when the exact scores are less important than the relative preferences or rankings. The *Pairwise Ranking Loss* is ideal for our needs:

$$L_{\text{Pairwise Ranking}} = -\log\left(\sigma(\boldsymbol{f}(g_1) - \boldsymbol{f}(g_2))\right) \tag{5}$$

where $\boldsymbol{f}$ represents the feature of the hardware IR graph, σ is the activation function, and g_1 and g_2 are logic equivalance graph being compared.

By employing ranking loss to train SimplifyGuider, the ranking of transformation options can be effectively learned, ensuring that beneficial transformations are prioritized.

F. Common Subexpression Elimination

The *SimplifyCFG* process may introduce common subexpressions, one such case being the example depicted in Fig. 2. To address this issue, the IR program can be traversed to identify duplicated operations. These duplicates are then safely removed without affecting the use and define dependencies within the IR program. This step is performed by the *Common Subexpression Elimination (CSE)* pass within MLIR.

V. Experimental Evaluation

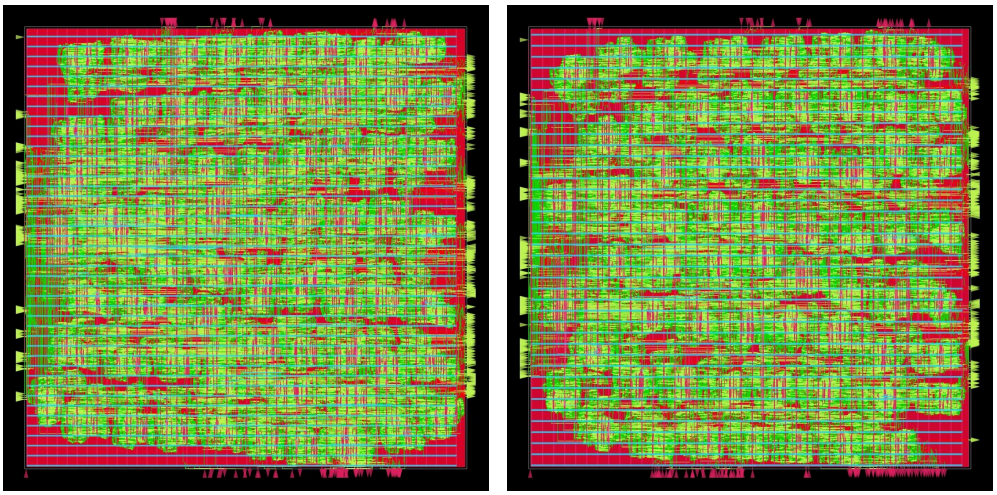
The proposed algorithm, CombRewriter, was implemented in C++ and integrated as a pass within CIRCT [16]. Comprehensive experiments were conducted to assess the performance of CombRewriter. These evaluations were performed on an 8-core Intel i7-11700 CPU @ 2.50GHz, within a software environment that included Chisel [1], CIRCT [16], Yosys [31], ABC [32].

A. Experimental Setup

Benchmark. To validate the performance of CombRewriter, a collection of arithmetic and conditional expressions representative of common hardware design patterns was constructed.

TABLE IV Evaluation Results: Case Study on *Gemmini* [18].

Method	**Standard Cells**	**Area** (μm^2)	**Clock Skew** (ps)	**Worst Slack** (ns)	**Route Length**
Original	200,178	23,861.50	51.94	0.795	730,358
CombRewriter	161,649	19,485.10	49.26	1.089	577,600
Improvement	19.25%	18.34%	5.16%	36.98%	20.92%



(a) P&R Result w/o CombRewriter. (b) P&R Result with CombRewriter.

Fig. 5 Visualization of the P&R Results on Design *Gemmini* [18] (Spatial 8 × 8).

These expressions match the original structures presented in TABLE I and TABLE II.

Baseline. E-morphic [23] was selected as the baseline for comparison due to its relevance to this study and its status as the current state-of-the-art (SOTA) approach in netlist optimization.

Evaluation Flow. Yosys was utilized as the Verilog front-end parser. Logic synthesis and technology mapping were performed using the command '`if -g; st; dch; ps; map; topo; upsize; dnsize; stime;`' in ABC with the *ASAP 7nm* technology library [33]. Equivalence of the combinational logic between the original and simplified netlists was verified using the `cec` command in ABC.

B. Evaluation Results

As shown in TABLE III, CombRewriter achieved superior improvements across all categories compared to E-morphic, except in the *Shift Operation Distribution* category, where E-morphic increased the delay. The superior performance can be attributed to the different levels of abstraction employed by each approach. CombRewriter operates at the high-level operation abstraction, while E-morphic works at the gate level. The optimization process involves extracting subgraphs to analyze potential simplifications. However, a high-level operation may require hundreds of gates to implement, resulting in prohibitively large subgraphs that make E-morphic optimization inefficient due to exponential search complexity.

C. Case Study

Experimental Settings. CombRewriter was further evaluated on the *Gemmini* [18] design, a machine learning accelerator, with Placement and Routing (P&R) performed to assess its impact on physical implementation. The open-source EDA tool OpenROAD [34] was employed, using the ASAP 7nm technology library [33].

Evaluation Results. The results are summarized in TABLE IV, with the P&R visualizations shown in Fig. 5. CombRewriter achieved improvements of *19.25%*, *18.34%*, *5.16%*, *36.98%*, and *20.92%* in *standard cell count*, *area*, *clock skew*, *worst slack*, and *route length*, respectively. As illustrated in Fig. 5(b), the layout generated from the netlist simplified by CombRewriter demonstrates a more uniform and less congested routing structure compared to Fig. 5(a), where no simplification was applied.

VI. Conclusion

This paper presents CombRewriter, an approach that applies compiler design principles to simplify combinational logic in MLIR-based hardware compilers. To exploit hardware IR-specific simplification opportunities, specific transformation are designed. To ensure transformations provide benefits, a machine learning-assisted performance analysis approach, SimplifyGuider, is tailored for hardware IR transformation. Experimental results demonstrate that CombRewriter efficiently and effectively simplifies combinational logic in hardware IR graph, resulting in netlist area reduction.

REFERENCES

[1] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing Hardware in a Scala Embedded Language," in *ACM/IEEE Design Automation Conference (DAC)*, 2012.

[2] C. Papon, "SpinalHDL: An Alternative Hardware Description Language," in *Free and Open source Software Developers' European Meeting (FOSDEM)*, 2017.

[3] J. Decaluwe, "MyHDL: A Python-Based Hardware Description Language," *Linux Journal*, 2004.

[4] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis *et al.*, "Spatial: A Language and Compiler for Application Accelerators," in *ACM SIGPLAN Symposium on Programming Language Design & Implementation (PLDI)*, 2018.

[5] R. Nigam, S. Thomas, Z. Li, and A. Sampson, "A Compiler Infrastructure for Accelerator Generators," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[6] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood, "A Pythonic Approach for Rapid Hardware Prototyping and Instrumentation," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, 2017.

[7] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson *et al.*, "Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017.

[8] S. Jiang, B. Ilbeyi, and C. Batten, "Mamba: Closing the Performance Gap in Productive Hardware Development Frameworks," in *ACM/IEEE Design Automation Conference (DAC)*, 2018.

[9] D. Lockhart, G. Zibrat, and C. Batten, "PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[10] K. Majumder and U. Bondhugula, "HIR: An MLIR-based Intermediate Representation for Hardware Accelerator Description," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[11] C. Mattarei, M. Mann, C. Barrett, R. G. Daly, D. Huff, and P. Hanrahan, "CoSA: Integrated Verification for Agile Hardware Design," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2018.

[12] R. Nikhil, "Bluespec SystemVerilog: Efficient, Correct RTL from High-Level Specifications," in *International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, 2004.

[13] F. Schuiki, A. Kurth, T. Grosser, and L. Benini, "LLHD: A Multi-level Intermediate Representation for Hardware Description Languages," in *ACM SIGPLAN Symposium on Programming Language Design & Implementation (PLDI)*, 2020.

[14] A. Sharifian, R. Hojabr, N. Rahimi, S. Liu, A. Guha, T. Nowatzki, and A. Shriraman, "μir-An Intermediate Representation for Transforming and Optimizing the Microarchitecture Of Application Accelerators," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[15] S.-H. Wang, R. T. Possignolo, H. B. Skinner, and J. Renau, "LiveHD: A Productive Live Hardware Development Flow," *IEEE Micro*, 2020.

[16] S. Eldridge, P. Barua, A. Chapyzhenka, A. Izraelevitz, J. Koenig, C. Lattner, A. Lenharth, G. Leontiev, F. Schuiki, R. Sunder *et al.*, "MLIR as Hardware Compiler Infrastructure," in *Workshop on Open-Source EDA Technology (WOSET)*, 2021.

[17] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," in *ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*, 2021.

[18] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao *et al.*, "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration," in *ACM/IEEE Design Automation Conference (DAC)*, 2021.

[19] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*, 2004.

[20] J. Cocke, "Global Common Subexpression Elimination," in *Proceedings of a Symposium on Compiler Optimization*, 1970.

[21] T. Lengauer and R. E. Tarjan, "A Fast Algorithm for Finding Dominators in a Flowgraph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1979.

[22] L. P. Cordella, P. Foggia, C. Sansone, M. Vento *et al.*, "An Improved Algorithm for Matching Large Graphs," in *IAPR-TC15 International Workshop on Graph-Based Representations in Pattern Recognition*, 2001.

[23] C. Chen, G. Hu, C. Yu, Y. Ma, and H. Zhang, "E-morphic: Scalable Equality Saturation for Structural Exploration in Logic Synthesis," in *ACM/IEEE Design Automation Conference (DAC)*, 2025.

[24] H. Wu, H. Zheng, Y. Pu, and B. Yu, "Circuit Representation Learning with Masked Gate Modeling and Verilog-AIG Alignment," in *International Conference on Learning Representations (ICLR)*, 2025.

[25] H. Zheng, H. Wu, Z. He, Y. z. Ma, and B. Yu, "iRw: An Intelligent Rewriting," in *IEEE/ACM Proceedings Design, Automation and Test in Eurpoe (DATE)*, 2025.

[26] H. Zheng, Z. He, F. Liu, Z. Pei, and B. Yu, "LSTP: A Logic Synthesis Timing Predictor," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, 2024.

[27] C. Xu, P. Sharma, T. Wang, and L. W. Wills, "Fast, Robust and Transferable Prediction for Hardware Logic Synthesis," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.

[28] Y. Zhang, H. Ren, and B. Khailany, "GRANNITE: Graph Neural Network Inference for Transferable Power Estimation," in *ACM/IEEE Design Automation Conference (DAC)*, 2020.

[29] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive Representation Learning On Large Graphs," in *Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.

[30] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender, "Learning to Rank Using Gradient Descent," in *International Conference on Machine Learning (ICML)*, 2005.

[31] C. Wolf, J. Glaser, and J. Kepler, "Yosys-A Free Verilog Synthesis Suite," in *IEEE Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[32] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool," in *International Conference on Computer-Aided Verification (CAV)*, 2010.

[33] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "ASAP7: A 7-nm finFET Predictive Process Design Kit," *Microelectronics Journal*, 2016.

[34] T. Ajayi, D. Blaauw, T. Chan, C. Cheng, V. Chhabria, D. Choo, M. Coltella, S. Dobre, R. Dreslinski, M. Fogaça *et al.*, "OpenROAD: Toward a Self-Driving, Open-Source Digital Layout Implementation Tool Chain," *Proceedings of Government Microcircuit Applications and Critical Technology Conference (GOMACTECH)*, 2019.