eLogic: An E-Graph-based Logic Rewriting Framework for Majority-Inverter Graphs

Rongliang Fu[1], Wei Xuan[2], Shuo Yin[1], Guangyu Hu[3], Chen Chen[3], Hongce Zhang[3], Bei Yu[1], Tsung-Yi Ho[1]
[1]CUHK [2]HKUST [3]HKUST-GZ
rlfu@cse.cuhk.edu.hk

***Abstract*—Majority-Inverter Graph (MIG) emerges as a promising data structure for logic optimization and synthesis, offering a more compact representation for logic functions compared to traditional AND/OR-Inverter graphs. Consequently, the MIG finds widespread application in digital circuit design, particularly in quantum circuits and superconducting adiabatic quantum-flux-parametron logic circuits. Currently, logic optimization techniques for MIG mainly fall into two categories: (i) logic rewriting with predefined more compact sub-structures and (ii) logic resubstitution with already existing logic in the Boolean network. However, the inherent complexity of MIG logic and the limitation imposed by the input scale of sub-structures significantly impact the performance of these methods. To address these challenges, this paper proposes eLogic, a novel depth-oriented MIG logic rewriting framework using e-graphs, to minimize the depth and size of MIG. The eLogic utilizes the e-graphs, a data structure for efficient computation with equalities between terms, to minimize the depth and size of the cone delimited by the cut. The experimental results on the EPFL benchmark demonstrate the effectiveness of eLogic. It is noteworthy that eLogic is open-sourced on https://github.com/Flians/eLogic.**

***Index Terms*—Logic synthesis, logic rewriting, majority-inverter graph, e-graph.**

I. Introduction

Logic synthesis is a crucial step in the design and implementation of digital circuits. It involves the translation of a high-level hardware description into a detailed and optimized gate-level representation. This process plays a significant role in modern digital design methodologies, as it enables designers to efficiently transform abstract design specifications into practical and efficient hardware implementations. Throughout this process, various presentation structures are commonly employed to represent and analyze digital circuits at different levels of abstraction, particularly at the Boolean network level. These structures are instrumental in facilitating the design, optimization, and verification of complex digital systems. Among them, the AND-Inverter graph (AIG) [1] is the most widely used logic representation in logic synthesis, alongside the Majority-Inverter graph (MIG) [2], the XOR-AND-Inverter graph (XAG) [3], and the XOR-Majority-Inverter graph (XMG) [4]. Recent studies [2], [5] show that the MIG not only encompasses the function represented by the AIG but also yields a more compact representation for a given logic function compared with the AIG. Therefore, MIG-based logic function designs have attracted more and more attention, particularly for emerging technologies and advanced computing paradigms [6]–[10].

Currently, many Boolean optimization techniques have been proposed for MIGs and can be summarized as two categories: i) Boolean resubstitution [11]–[13] and ii) Boolean rewriting [14], [15]. Boolean resubstitution techniques focus on optimizing a specific node by replacing it with a new function derived from its local neighborhood. However, these techniques are limited in their ability to explore the solution space, as they only consider functions that can be formed using the existing nodes in the local neighborhood. On the other hand, Boolean rewriting techniques utilize a set of predefined rewriting rules to transform subgraphs into more efficient representations. These techniques can explore a broader solution space by applying various rewriting rules, but for MIGs, they typically require a large cut size to achieve significant optimization. For example, as shown in Fig. 1, optimizing the depth of node f requires a 6-cut (Fig. 1(a)), while optimizing the size of node f requires a 5-cut (Fig. 1(b)). The need for large cut sizes can lead to increased computational complexity and longer optimization times, especially in state-of-the-art rewriting techniques using exact synthesis [14], [15].

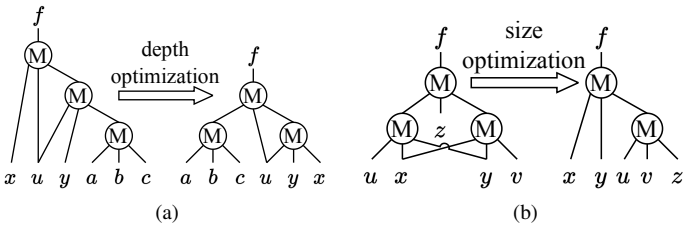

Fig. 1. MIG logic optimization usually requires a large cut size. (a) shows the depth optimization for a 6-cut of node f. (b) shows the size optimization for a 5-cut of node f. (M) represents a node with a 3-input majority function.

To address these challenges, we propose **eLogic**, an e-graph-based logic rewriting framework for MIGs. The e-graph [16], [17] is a data structure that efficiently represents a large number of equivalent expressions by grouping them into equivalence classes. By systematically applying rewrite rules until saturation, e-graphs naturally handle the complex multi-variable MIG transformations while avoiding the exponential search complexity faced by traditional exact synthesis methods on large cuts. This enables eLogic to achieve improved optimization results in terms of both depth and size reduction for MIGs.

Overall, the main contributions of this work are as follows:

- We introduce eLogic, an open-source MIG logic rewriting framework based on e-graphs, which enables efficient and effective MIG logic optimization.
- We develop an intermediate language to represent MIG functions in the e-graph structure, along with rewriting rules to explore circuit topologies with larger cut sizes.

- We combine e-graph rewriting on large cuts with exact synthesis using don't cares on small cuts to optimize both depth and size.
- We evaluate eLogic on the EPFL benchmark and demonstrate that it outperforms existing state-of-the-art MIG optimization techniques in both depth and size.

II. Preliminaries

A. Majority-Inverter Graph

As a directed acyclic graph, the MIG is a logic representation structure composed of nodes with the majority function and regular/complemented edges indicating the presence/absence of an inverter ($\neg$). The majority function produces an output based on the majority of its inputs. The node usually has three inputs and operates at a three-input majority function, denoted as $M(x, y, z) = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$. Since the majority function can represent the AND ($\wedge$) logic operation, such as $x \wedge y = M(x, 0, y)$, the MIG can extend the representation capabilities of the AIG, *i.e.*, MIG $\supset$ AIG [5]. Equation (1) shows the five transformations [5], which form a sound and complete axiomatic system for the MIG-based Boolean algebra.

$$\Omega \begin{cases} \textbf{Commutativity}(\boldsymbol{\Omega.C}): \\ M(x,y,z) = M(y,x,z) = M(z,y,x) \\ \textbf{Majority}(\boldsymbol{\Omega.M}): \\ \begin{cases} \text{if}(x = y): M(x,x,z) = M(y,y,z) = x = y \\ \text{if}(x = \neg y): M(x,\neg x,z) = M(\neg y,y,z) = z \end{cases} \\ \textbf{Associativity}(\boldsymbol{\Omega.A}): \\ M\left(x,u,M(y,u,z)\right) = M\left(z,u,M(y,u,x)\right) \\ \textbf{Distributivity}(\boldsymbol{\Omega.D}): \\ M\left(x,y,M(u,v,z)\right) = M\left(M(x,y,u),M(x,y,v),z\right) \\ \textbf{Inverter Propagation}(\boldsymbol{\Omega.I}): \\ \neg M(x,y,z) = M(\neg x,\neg y,\neg z) \end{cases} \tag{1}$$

These axioms often involve a large number of input variables, especially for **Distributivity** with 5 variables, which poses significant challenges for traditional exact synthesis methods due to the exponential number of potential equivalent expressions that must be explored. Furthermore, based on these axioms, various theorems can be derived to facilitate MIG manipulation. For instance, Fig. 2 illustrates the derivation of the **Complementary Associativity** theorem ($\boldsymbol{\Psi.C}$):

$$M\left(x,u,M(y,\neg u,z)\right) = M\left(x,u,M(y,x,z)\right), \tag{2}$$

which is not included in the original axioms but is useful for MIG transformations. These axioms and derived theorems enable systematic manipulation and optimization of MIGs through a series of transformations.

B. Logic Rewriting

Several established logic-rewriting frameworks have demonstrated strong optimization capabilities. Early approaches include map-based logic optimization techniques [18] and DAG-aware logic rewriting [19]. More recently, heuristic logic resubstitution [13] and Boolean matching with don't cares [15] have achieved substantial reductions in circuit size. Among these, logic rewriting with don't cares has emerged as a particularly effective technique for MIG optimization.

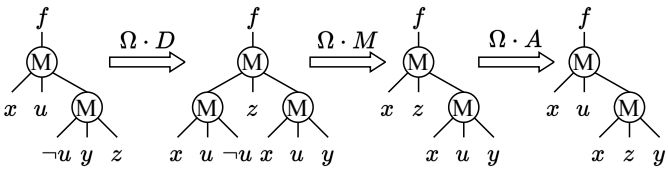

Fig. 2. The derivation of complementary associativity.

In general, logic rewriting with don't cares proceeds in three steps: (i) partitioning a Boolean network into subsets by constructing multiple cut sets for each node, (ii) computing don't cares for each cut, and (iii) performing Boolean matching with these don't cares. However, this method faces two key challenges. First, computing don't-cares per cut is computationally expensive and scales poorly with cut size. Second, Boolean matching relies on a predefined library obtained via exact synthesis, which is itself limited by cut size. Consequently, the effectiveness of current methods is fundamentally constrained by cut size, motivating the need for a rewriting method that can efficiently handle larger cuts, which is the focus of this work.

C. E-Graphs

Equality saturation [17] is a program optimization technique that systematically explores a space of equivalent programs by applying rewrite rules to generate semantically equivalent expressions. Unlike traditional methods, which apply transformations sequentially, equality saturation uses an *e-graph* [16] to represent and manipulate a large set of equivalent expressions simultaneously. This enables exhaustive exploration of the optimization space, often yielding better results.

The e-graph is a compact data structure that represents equivalence classes of expressions, where nodes correspond to expressions and edges denote equivalences. By sharing common sub-expressions and supporting efficient union-find operations, e-graphs reduce redundancy and memory overhead while maintaining a rich set of equivalent expressions. The equality saturation process consists of two phases: (1) *saturation*, where rewrite rules are applied exhaustively to grow the e-graph, and (2) *extraction*, where the optimal program is selected by assigning costs to expressions and choosing the lowest-cost representative.

A key advancement in this field is *egg* [20], a highly optimized toolkit for equality saturation. The *egg* provides a flexible framework for defining rewrite rules, performing saturation, and extracting optimized programs. Its efficiency and scalability have made it a popular choice for tasks such as compiler optimizations [21], algebraic simplification [22], and EDA, involving high-level synthesis (HLS) [23]–[25] and AIG-based logic optimization [26]. Hence, this paper leverages the capabilities of e-graphs and *egg* to address the challenges of MIG logic rewriting, enabling efficient exploration of large cut sizes and complex transformations.

III. E-Graph Construction for MIG Rewriting

A. Intermediate Language for MIG Representation

To leverage the e-graph structure for MIG rewriting, we first need to define an intermediate language that can accurately

represent MIG functions within the e-graph. This language should capture the essential elements of MIGs, including majority nodes and inverter edges, while also being compatible with the e-graph's capabilities for representing equivalence classes. We define the intermediate language using a set of operators and constructs that correspond to the components of an MIG. The primary operator is the majority function, denoted as 'M', which takes three inputs. The inverter is represented using a unary operator, denoted as '¬', which can be applied to any input or output of the majority function. Additionally, we include constructs for primary inputs and constants (0 and 1) to complete the representation. The syntax of the intermediate language can be formally defined as follows:

- **Primary Inputs (PI)**: The input variables of the MIG.
- **Constants**: The binary constants 0 and 1.
- **Expressions**: $E ::= (M\ E\ E\ E) \mid (\neg\ E) \mid pi \in \textbf{PI} \mid 0 \mid 1.$

This intermediate language allows us to construct e-graphs that accurately reflect the structure and functionality of MIGs. Each expression in the language corresponds to a node or subgraph in the MIG, enabling us to perform rewriting operations directly on the e-graph.

B. Rewriting Rules for MIG Rewriting

To effectively utilize the e-graph for MIG rewriting, we need to define a comprehensive set of rewriting rules that can transform expressions in our intermediate language while preserving their semantics. These rules should be derived from the axioms and theorems of MIGs, as well as additional transformations that facilitate optimization. The rewriting rules can be categorized into several types:

- **Axiomatic Rules**: These rules are directly derived from the five MIG axioms shown in Equation (1). The Commutativity ($\boldsymbol{\Omega.C}$), Associativity ($\boldsymbol{\Omega.A}$), Distributivity ($\boldsymbol{\Omega.D}$), and Inverter Propagation ($\boldsymbol{\Omega.I}$) axioms are translated into bidirectional rewrite rules to enable systematic exploration of equivalent MIG representations. The Majority axiom ($\boldsymbol{\Omega.M}$) provides unidirectional simplification rules:
$$\begin{aligned}(M\ x\ x\ z) &\Rightarrow x\\ (M\ x\ (\neg\ x)\ z) &\Rightarrow z\end{aligned} \tag{3}$$
which reduce expressions when duplicate or complementary inputs are detected, but cannot be applied in reverse as this would introduce new variables.
- **Derived Theorems**: These rules are based on theorems that can be derived from the axioms, including
  - Complementary Associativity:
$$(M\ x\ u\ (M\ y\ (\neg\ u)\ z)) \Leftrightarrow (M\ x\ u\ (M\ y\ x\ z)) \tag{4}$$
  - Complementary Majority:
$$(M\ x\ y\ (\neg\ y)) \Leftrightarrow (M\ x\ y\ x) \tag{5}$$
  - XNOR $y \odot z = M(1, M(0, y, z), \neg M(1, y, z))$:
$$\begin{aligned}&(M\ x\ (M\ 0\ y\ z)\ (\neg\ (M\ 1\ y\ z))) \Leftrightarrow\\ (M\ 0\ x\quad &(M\ 1\ (M\ 0\ y\ z)\ (\neg\ (M\ 1\ y\ z))))\end{aligned} \tag{6}$$
  - XOR $y \oplus z = M(0, \neg M(0, y, z), M(1, y, z))$:
$$\begin{aligned}&(M\ x\ (\neg\ (M\ 0\ y\ z))\ (M\ 1\ y\ z)) \Leftrightarrow\\ (M\ 1\ x\quad &(M\ 0\ (\neg\ (M\ 0\ y\ z))\ (M\ 1\ y\ z)))\end{aligned} \tag{7}$$
- **Optimization Rules**: These rules involve transformations of constants and primary inputs, including
  - Constants: $(\neg\ 0) \Leftrightarrow 1$ and $(\neg\ 1) \Leftrightarrow 0$
  - Primary Inputs: $(\neg\ (\neg\ x)) \Leftrightarrow x$

By applying these rewriting rules within the e-graph, we can systematically explore a wide range of equivalent expressions for a given MIG function. The e-graph structure allows us to efficiently manage and combine these expressions, enabling effective optimization through equality saturation.

IV. MIG Rewriting

After constructing the e-graph with the defined intermediate language and rewriting rules, we can proceed to the MIG rewriting. This section outlines the process of rewriting MIGs using the e-graph, focusing on the extraction of optimized representations based on a defined cost model.

A. E-Graph-based Subgraph Rewriting

The e-graph-based subgraph rewriting process involves utilizing the e-graph to explore and identify optimal representations of a given MIG function. This process involves two main phases: saturation and extraction. The saturation phase involves iteratively applying the rewriting rules defined in Section III-B to the e-graph until no new equivalence classes can be formed. This process ensures that all possible equivalent expressions for the MIG function are represented within the e-graph. The saturation process is implemented using a worklist algorithm, where each time a rule is applied, the resulting expressions are added to the worklist for further processing. The saturation continues until the worklist is empty, indicating that no further transformations are possible.

Once the e-graph is fully saturated, we proceed to the extraction phase. To guide the extraction of optimized MIG representations from the e-graph, we need to define a cost model that evaluates the quality of different expressions based on their depth and size. The cost model $C(E)$ of an expression E is defined as a tuple $\{d, m, n\}$, including the following parts:

- **Depth Cost**: The depth of an expression is defined as the maximum number of majority nodes on any path from an input to the output. The depth cost can be calculated as:
$$d(E) = \text{max depth of majority nodes in } E. \tag{8}$$
- **Size Cost**: The size of an expression is defined as the total number of majority nodes in the expression. The size cost can be calculated as:
$$m(E) = \text{number of majority nodes in } E. \tag{9}$$
- **Inverter Cost**: The inverter count of an expression is defined as the total number of complemented edges in the expression. The inverter cost can be calculated as:
$$n(E) = \text{number of complemented edges in } E. \tag{10}$$

Specifically, we define the cost of each element in our defined intermediate language as follows:

- **Primary Input** $pi \in$ **PI**: $C(pi) = \{d(pi), 0, 0\}$, where $d(pi)$ is the initial depth of given input pi.
- **Constants**: $C(0) = \{0, 0, 0\}$ and $C(1) = \{0, 0, 0\}$.
- **Majority**: For an expression $E = (M\ E_1\ E_2\ E_3)$, the cost is defined as:

$$\begin{aligned} C(E) = \{&1 + \max\left(d(E_1), d(E_2), d(E_3)\right), \\ &1 + m(E_1) + m(E_2) + m(E_3), \\ &n(E_1) + n(E_2) + n(E_3)\}, \end{aligned} \tag{11}$$

  where the depth and size both increase by 1 due to the addition of the majority node.
- **Inverter**: For an expression $E = (\neg\ E_1)$, the cost is defined as:

$$C(E) = \{d(E_1), m(E_1), 1 + n(E_1)\}. \tag{12}$$

  Here, the addition of the inverter does not affect the depth and size, but the number of inverters increases by 1.

To compare and select the most efficient representation, we define the addition and comparison operations for the cost tuples as follows:

- **Addition**: The addition of two cost tuples $C_1 = \{d_1, m_1, n_1\}$ and $C_2 = \{d_2, m_2, n_2\}$ is defined as:

$$C_1 + C_2 = \{\max\left(d_1, d_2\right), m_1 + m_2, n_1 + n_2\}. \tag{13}$$

- **Comparison**: The comparison of two cost tuples $C_1 = \{d_1, m_1, n_1\}$ and $C_2 = \{d_2, m_2, n_2\}$ is defined lexicographically, prioritizing depth, then size, and finally inverter count:

$$\begin{aligned} C_1 < C_2 \text{ if } &(d_1 < d_2) \vee \\ &(d_1 = d_2 \wedge m_1 < m_2) \vee \\ &(d_1 = d_2 \wedge m_1 = m_2 \wedge n_1 < n_2). \end{aligned} \tag{14}$$

During the extraction phase, we employ the default extractor provided by *egg* [20]. This extractor adopts a dynamic programming approach, iteratively calculating the minimum cost for each equivalence class to identify the optimal representative. Consequently, the selected expression minimizes depth first, then size, and finally inverter count according to the lexicographic ordering defined above. In this way, we implement an e-graph-based MIG rewriting engine that enables us to systematically explore and identify optimal MIG representations.

B. eLogic Framework

The proposed eLogic framework integrates two complementary methodologies: e-graph-based rewriting for handling large cuts and exact synthesis with don't cares for small cuts, as discussed in Section II-B. This hybrid approach leverages the scalability of e-graph equality saturation for exploring extensive search spaces while exploiting the optimality of exact synthesis for compact functions.

Algorithm 1 summarizes the comprehensive procedure of our eLogic framework. The procedure iterates over all nodes in topological order from primary inputs to primary outputs (lines 1-24), thereby ensuring adherence to dependency constraints while enabling incremental updates. For each node v, the procedure first calculates its current level (line 2) and determines whether it is on the critical path (line 3) to guide subsequent cut

Algorithm 1: The eLogic algorithm.

Input: MIG $N(V, E)$, cut size K, window size L, threshold λ.
Output: Optimized MIG.

```
1  for node v ∈ V in topological order do
2      original ← calculate the level of node v;
3      critical ← determine if node v is on a critical path;
4      W ← construct a L-input reconvergent window for node v;
5      G ← ∅, best ← {0, original};
6      for C ← enumerate K-cuts of node v do
7          if |C| > λ then
8              E ← describe cut C using the intermediate language;
9              Es ← perform e-graph-based rewriting for E
                prioritizing depth if critical else size;
10             Gs ← reconstruct MIGs for Es;
11         else
12             S ← ∅, dc ← ∅;
13             if C is contained in W then
14                 S ← incrementally simulate the window W;
15                 dc ← calculate don't cares for cut C on window
                    W using S;
16             Gs ← perform Boolean matching for cut C with dc
                and its truth table in the exact database;
17         for G' ∈ Gs do
18             cur ← calculate the gain from replacing C with G';
19             if ❶(critical ∧ cur.n > best.n ∧ cur.l ≤ best.l)∨
20               ❷(¬critical ∧ cur.n > best.n)∨
21               ❸(cur.n = best.n ∧ cur.l < best.l) then
22                 G ← G', best ← cur;
23     if best.n > 0 ∨ (best.n = 0 ∧ best.l < original) then
24         N ← replace C with G in network N;
25 return N.
```

rewriting. A L-input reconvergent window is then constructed for node v (line 4) to localize the computation of don't cares. Subsequently, the K-cuts of node v are enumerated (lines 6-22) to identify potential candidates for rewriting.

Specifically, the procedure traverses each K-cut $\mathcal{C}$ of node v. Given that exact synthesis using don't cares can handle small cuts efficiently, a threshold λ is set to determine the appropriate approach for each cut (line 7). If the cut size exceeds λ, e-graph-based rewriting is performed (lines 8-10) to explore a broader range of equivalent representations. The cut is described using the intermediate language proposed in Section III-A (line 8), and e-graph-based rewriting is applied to generate alternative structures (line 9). During this process, the optimization priority is dynamically adjusted: depth is prioritized if the node is on the critical path; otherwise, size is prioritized. The resulting expressions are then reconstructed into MIGs Gs (line 10). If the cut size is less than λ, exact synthesis with don't cares is employed. Initially, the don't cares are calculated when cut $\mathcal{C}$ is contained in the window $\mathcal{W}$ (lines 13-15). Subsequently, Boolean matching is performed in the exact database to identify alternative structures Gs (line 16).

For each candidate structure $G' \in Gs$, the gain cur from replacing $\mathcal{C}$ with G' is calculated (line 18). The gain includes two parts: {the number n of reduced nodes, the level l of node v after replacement}. The replacement selection follows a priority-based strategy with three criteria: ❶ if node v is on

TABLE I. Experimental results of the single methods on the EPFL benchmark [27].

Circuit	Original		resub [13]		rw [15]		eLogic	
	size	depth	size	depth	size	depth	size	depth
adder	1020	255	893	129	516	130	516	130
bar	3336	12	3208	13	3010	13	3141	12
div	57247	4372	52216	4341	46360	4308	26225	2194
hyp	214335	24801	199058	9291	156590	9155	140037	8923
log2	32060	444	32005	444	29312	421	27659	243
max	2865	287	2837	287	2356	208	2444	133
multiplier	27062	274	26802	273	24370	272	25082	174
sin	5416	225	5331	227	4934	195	4864	130
sqrt	24618	5058	20801	5882	18813	5956	22573	4086
square	18484	250	18089	164	16392	134	16824	128
arbiter	11839	87	11711	87	11839	87	5511	35
cavlc	693	16	651	16	598	15	663	11
ctrl	174	10	101	9	108	7	127	6
dec	304	3	304	3	304	3	304	3
i2c	1342	20	1247	20	1203	17	1259	11
int2float	260	16	234	16	210	14	239	10
mem_ctrl	46836	114	45349	115	41180	113	44265	94
priority	978	250	817	240	875	241	898	155
router	257	54	257	54	242	52	250	32
voter	13758	70	9372	68	8094	63	7884	57
Ave. ratio	1.00	1.00	0.91	0.93	0.83	0.87	**0.82**	**0.63**

a critical path, accept when more nodes are reduced without increasing delay; ❷ if node v is not on a critical path, accept when nodes are reduced; ❸ when node reduction is equal, accept if delay is improved. If any criterion is satisfied, the best candidate and its gain are updated (lines 19–22).

If the best candidate achieves improvement, the replacement is applied to the network (line 24). The algorithm iterates through all nodes and returns the optimized network (line 25).

V. Experimental Results

The proposed eLogic framework was implemented using Rust and C++. The e-graph-based MIG rewriting engine was developed using the *egg* toolkit [20]. Besides, eLogic employed the *mockturtle* [28], an open-source logic network library, to construct the exact library and implement Boolean matching with don't cares. All experiments were conducted on a machine running Ubuntu 22.04 and equipped with Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz and 256.0 GB of memory. For parameter configuration, we set the cut size $K = 8$, the window size $L = 12$, and the threshold $\lambda = 4$.

We evaluated eLogic on the EPFL benchmark suite [27], which comprises 20 circuits spanning arithmetic, signal processing, and control logic. The circuits are originally described in AIGER format and synthesized into MIGs using mockturtle's `aiger_reader` command. For comparison, two state-of-the-art baselines were selected: (i) "resub": logic resubstitution with don't cares [13], and (ii) "rw": exact logic rewriting with don't cares [15]. Both baselines employed a cut size of $K = 4$ and a window size of $L = 12$, and all results were validated for functional equivalence.

TABLE I presents the experimental results of single methods on MIG optimization. The "Original" part shows the initial attributes of input circuits, including "size", the original number of majority nodes in the circuit, and "depth", the original circuit depth. The "resub" and "rw" parts show the outcomes of the respective baselines, while the "eLogic" part presents the results of our approach. Across all benchmarks, eLogic achieves substantial improvements, reducing the size by 18.27% and circuit depth by 36.84% on average compared to the original circuits. Relative to "resub", eLogic delivers an average size reduction of 9.91% and depth reduction of 30.41%. Compared to "rw", eLogic achieves a 1.16% average size reduction and a 25.40% average depth reduction.

Furthermore, we evaluated the performance of combined methods, where eLogic was applied before the baselines. TABLE II presents the experimental results of the combined methods on the EPFL benchmark. The "resub+resub" and "rw+rw" parts represent the results of applying the same baseline twice, while the "eLogic+resub" and "eLogic+rw" parts show the results of applying eLogic followed by the respective baseline. The "time(s)" part shows the corresponding runtime in seconds of each method. Experimental results demonstrate that combining eLogic with the baselines leads to further improvements while maintaining efficiency. When compared with applying the logic resubstitution method [13] twice, combining eLogic with resubstitution achieves an average size reduction of 15.12% and an average depth reduction of 22.53%. In comparison with applying the exact logic rewriting method [15] twice, combining eLogic with rewriting achieves an average size reduction of 4.26% and an average depth reduction of 20.17%.

In terms of runtime efficiency, eLogic is slightly slower. This is because the e-graph-based rewriting engine starts from scratch for each subgraph input. This means that every time a new subgraph is considered, the e-graph is rebuilt and all rewrite rules are reapplied from the beginning. Unlike traditional methods that reuse previous computations or incrementally update the graph, eLogic does not retain any intermediate optimization state between subgraphs. This repeated initialization and exhaustive rewriting for each subgraph increase computational overhead. To address this issue, we also construct a library to store all optimized MIGs, thereby speeding up the rewriting process. As a result, eLogic can maintain reasonable efficiency and complete optimization for all benchmarks within 30 minutes.

VI. Conclusion

This paper presented **eLogic**, an e-graph-based logic rewriting framework for MIGs. In eLogic, e-graph-based logic rewriting is applied to large cuts, while exact synthesis using don't cares is employed for small cuts. This hybrid approach enables efficient and effective optimization of both circuit depth and size. We developed an intermediate language and a comprehensive set of rewriting rules to systematically explore the solution space, and proposed a cost model to guide the extraction of optimal MIG representations. Experimental results on the EPFL benchmark suite demonstrate that eLogic consistently outperforms state-of-the-art MIG optimization techniques, achieving significant reductions in both node count and circuit depth. Furthermore, combining eLogic with existing methods yields further improvements, highlighting its versatility and practical value for digital circuit synthesis.

TABLE II. Experimental results of the combined methods on the EPFL benchmark [27].

Circuit	resub [13]+resub [13]			rw [15]+rw [15]			eLogic+resub [13]			eLogic+rw [15]		
	size	depth	time(s)	size	depth	time(s)	size	depth	time(s)	size	depth	time(s)
adder	893	129	0.23	514	130	0.03	515	130	0.13	514	130	0.07
bar	3208	13	0.64	3010	13	0.26	3007	13	1.12	2981	13	0.90
div	51590	4145	29.49	38210	3020	8.95	25955	2182	223.46	24575	2185	218.21
hyp	199058	9291	200.12	146181	9123	67.33	133608	13891	1234.78	129116	8950	1192.09
log2	32001	444	12.99	29119	421	4.73	26447	265	1421.78	25264	226	1416.43
max	2837	287	0.66	2320	208	0.22	2408	134	4.71	2342	138	4.41
multiplier	26801	273	8.68	24106	273	3.78	24708	179	216.41	20383	167	212.91
sin	5325	227	1.80	4877	195	0.79	4635	142	5.55	4412	130	4.71
sqrt	18568	2892	9.20	11955	2529	3.65	18934	4015	70.21	17383	3995	67.02
square	18088	164	4.01	16105	134	2.43	15149	130	18.43	12913	130	17.00
arbiter	11711	87	3.38	11839	87	0.79	4301	23	4.89	5291	36	4.21
cavlc	648	16	0.10	598	15	0.04	611	11	0.18	619	11	0.15
ctrl	98	9	0.03	104	7	0.01	102	7	0.03	112	5	0.03
dec	304	3	0.06	304	3	0.05	304	3	0.06	304	3	0.05
i2c	1236	20	0.16	1195	17	0.08	1187	11	0.28	1204	11	0.25
int2float	230	16	0.04	208	14	0.02	226	11	5.05	222	10	5.05
mem_ctrl	45172	115	19.79	40985	113	4.60	42427	93	428.69	41876	94	419.87
priority	817	240	0.26	863	241	0.10	816	155	0.36	835	156	0.24
router	257	54	0.08	242	52	0.03	246	32	0.08	239	32	0.06
voter	8959	67	2.80	5560	59	1.44	6926	62	11.27	6326	54	10.39
Ave. ratio	0.91	0.90	0.62	0.79	0.82	0.22	0.76	0.66	1.16	**0.74**	**0.63**	1.00

* Ave. ratios of "size" and "depth" are relative to the "Original" in TABLE I.

ACKNOWLEDGMENT

The research work described in this paper was conducted in the JC STEM Lab of Intelligent Design Automation funded by The Hong Kong Jockey Club Charities Trust and was supported in part by the Research Grants Council of Hong Kong SAR (Grant No. CUHK14207523); in part by the Research Grants Council of Hong Kong SAR (Grant No. CUHK14208021).

REFERENCES

[1] L. Hellerman, "A catalog of three-variable Or-Invert and And-Invert logical circuits," *IEEE Transactions on Electronic Computers*, vol. EC-12, no. 3, pp. 198–223, 1963.

[2] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-Inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *Proc. DAC*, 2014, pp. 1–6.

[3] G. Meuli, M. Soeken, and G. Micheli, "Xor-And-Inverter graphs for quantum compilation," *npj Quantum Information*, vol. 8, 12 2022.

[4] W. Haaswijk, M. Soeken, L. Amarù, P.-E. Gaillardon, and G. De Micheli, "A novel basis for logic rewriting," in *Proc. ASPDAC*, 2017, pp. 151–156.

[5] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-Inverter graph: A new paradigm for logic optimization," *IEEE TCAD*, vol. 35, no. 5, pp. 806–819, 2016.

[6] T. Zhang, H. Jiang, H. Mo, W. Liu, F. Lombardi, L. Liu, and J. Han, "Design of majority logic-based approximate booth multipliers for error-tolerant applications," *IEEE TNANO*, vol. 21, pp. 81–89, 2022.

[7] G. Meuli, V. Possani, R. Singh, S.-Y. Lee, A. T. Calvino, D. S. Marakkalage, P. Vuillod, L. Amaru, S. Chase, J. Kawa, and G. De Micheli, "Majority-based design flow for AQFP superconducting family," in *Proc. DATE*, 2022, pp. 34–39.

[8] R. Fu, J. Huang, M. Wang, Y. Nobuyuki, B. Yu, T.-Y. Ho, and O. Chen, "BOMIG: A majority logic synthesis framework for AQFP logic," in *Proc. DATE*, 2023, pp. 1–2.

[9] A. Deb, K. Datta, M. Hassan, S. Shirinzadeh, and R. Drechsler, "Automated equivalence checking method for majority based in-memory computing on ReRAM crossbars," in *Proc. ASPDAC*, 2023, pp. 19–25.

[10] R. Fu, O. Chen, N. Yoshikawa, and T.-Y. Ho, "Exact logic synthesis for reversible quantum-flux-parametron logic," in *ICCAD*, 2023, pp. 1–9.

[11] H. Riener, E. Testa, L. Amaru, M. Soeken, and G. De Micheli, "Size optimization of MIGs with an application to QCA and STMG technologies," in *Proc. NANOARCH*, 2018, pp. 157–162.

[12] S.-Y. Lee, H. Riener, A. Mishchenko, R. K. Brayton, and G. De Micheli, "A simulation-guided paradigm for logic synthesis and verification," *IEEE TCAD*, vol. 41, no. 8, pp. 2573–2586, 2022.

[13] S.-Y. Lee and G. D. Micheli, "Heuristic logic resynthesis algorithms at the core of peephole optimization," *IEEE TCAD*, vol. 42, no. 11, pp. 3958–3971, 2023.

[14] H. Riener, W. Haaswijk, A. Mishchenko, G. Micheli, and M. Soeken, "On-the-fly and DAG-aware: Rewriting boolean networks with exact synthesis," in *Proc. DATE*, 03 2019, pp. 1649–1654.

[15] A. T. Calvino and G. De Micheli, "Scalable logic rewriting using don't cares," in *Proc. DATE*, 2024, pp. 1–6.

[16] C. G. Nelson, "Techniques for program verification," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 1980, aAI8011683.

[17] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: a new approach to optimization," in *Proc. POPL*, 2009, p. 264–276.

[18] A. T. Calvino, H. Riener, S. Rai, A. Kumar, and G. De Micheli, "A versatile mapping approach for technology mapping and graph optimization," in *Proc. ASPDAC*, 2022, pp. 410–416.

[19] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: a fresh look at combinational logic synthesis," in *Proc. DAC*, 2006, pp. 532–535.

[20] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "egg: Fast and extensible equality saturation," *Proc. PACMPL*, vol. 5, no. POPL, pp. 1–29, 2021.

[21] S. Thomas and J. Bornholt, "Automatic generation of vectorizing compilers for customizable digital signal processors," in *Proc. ASPLOS*, 2024, pp. 19–34.

[22] Y. R. Wang, S. Hutchison, J. Leang, B. Howe, and D. Suciu, "SPORES: sum-product optimization via relational equality saturation for large scale linear algebra," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 1919–1932, 2020.

[23] E. Ustun, I. San, J. Yin, C. Yu, and Z. Zhang, "IMpress: Large integer multiplication expression rewriting for FPGA HLS," in *Proc. FCCM*, 2022, pp. 1–10.

[24] J. Cheng, S. Coward, L. Chelini, R. Barbalho, and T. Drane, "SEER: Super-optimization explorer for high-level synthesis using e-graph rewriting," in *Proc. ASPLOS*, 2024, pp. 1029–1044.

[25] S. Coward, T. Drane, and G. A. Constantinides, "ROVER: RTL optimization via verified e-graph rewriting," *IEEE TCAD*, vol. 43, no. 12, pp. 4687–4700, 2024.

[26] C. Chen, G. Hu, D. Zuo, C. Yu, Y. Ma, and H. Zhang, "E-Syn: E-graph rewriting with technology-aware cost functions for logic synthesis," in *Proc. DAC*, 2024.

[27] L. Amarù, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *Proc. IWLS*, 2015.

[28] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, S.-Y. Lee, A. Tempia Calvino, and G. Marakkalage, Dewmini Sudara De Micheli, "The EPFL logic synthesis libraries," 2022.