A Database Dependent Framework for K-Input Maximum Fanout-Free Window Rewriting

Xuliang Zhu[1,*], Ruofei Tang[1,*], Lei Chen[2,†], Xing Li[2], Xin Huang[1], Mingxuan Yuan[2], Weihua Sheng[3], Jianliang Xu[1]
[1] Department of Computer Science, Hong Kong Baptist University, Hong Kong, China
[2] Huawei Noah's Ark Lab, Hong Kong, China
[3] Huawei Hong Kong Research Center, Hong Kong, China
{csxlzhu, csrftang, xinhuang, xujl}@comp.hkbu.edu.hk, {lc.leichen, li.xing3, yuan.mingxuan, sheng.weihua}@huawei.com

***Abstract*—Rewriting is a widely used logic optimization approach incorporated in most commercial logic synthesis tools. In this paper, we present a new rewriting method based on And-Inverted Graph (AIG). Rather than focusing on cut rewriting, it considers a novel sub-structure called Maximum Fanout-Free Window (MFFW) and rewrites with a more compact implementation. Both exact synthesis and heuristic methods can be adopted to optimize MFFWs. A database dependent framework is proposed to store the optimal sub-structures to accelerate the processing. We further propose the semi-canonicalization to reduce the scale of the database, which could reduce more than 98% of the 4-input MFFW database. Extensive experiments on benchmark datasets demonstrate both the effectiveness and efficiency of our proposed framework.**

I. Introduction

Logic optimization is a process of finding an equivalent but more compact representation of a multi-level Boolean network. It plays a key role for the area and delay optimization in automated design flow. It is usually applied to a technology-independent representation of the network parsed from high-level descriptions. And-Inverter Graph (AIG), which is composed of two-input And gates and Inverters, is such a modern logic representation that has been widely adopted in logic optimization.

Due to the NP-hardness of logic optimization, different algorithms based on heuristics or local transformations have been proposed towards the optimization in different metrics. Among them, cut rewriting [1], [2], [3], [4], [5] is a powerful and widely used area-oriented optimization method that iteratively selects and greedily replaces small-scale sub-graphs with more compact structures. A pre-computed or runtime database is used to store the optimum implementations and avoid redundant calculations. A common sense under such a setting is that, larger considered sub-graphs for rewriting has more potential in creating better optimized networks. However, a k-cut is a single-output structure with k-inputs and only small-scale vertices are allowed to optimize. See Fig. 2 as an example. Fig. 2(a) shows a real sub-structure of the DIV circuit in the EPFL benchmark. After enumerating all cuts in the example, cut rewriting can only reduce one vertex 8 as shown in Fig. 2(b). If we extend the cut into a fully multiple-outputs structure, we can find a better implementation in Fig. 2(c), which is equivalent with Fig. 2(a) but can reduce three vertices.

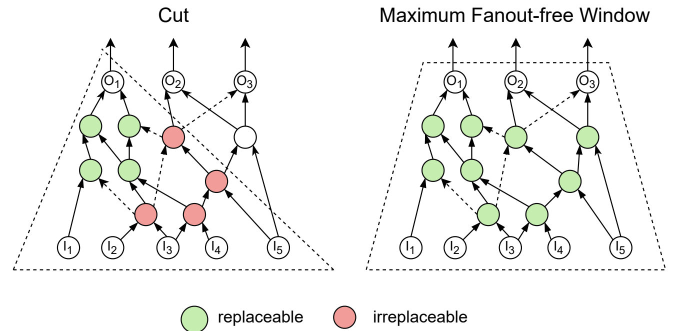
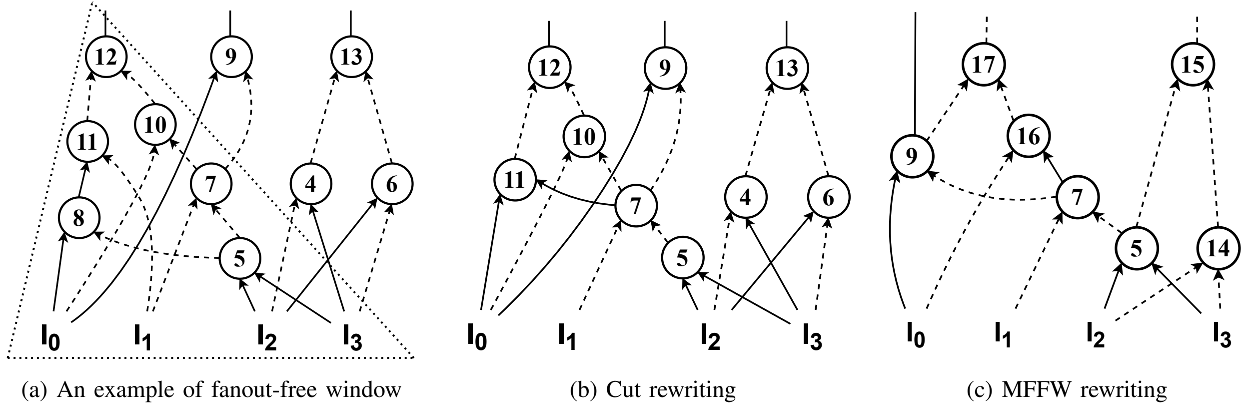


Fig. 1: Comparison of cut and MFFW. The green vertices are replaceable and red vertices are irreplaceable. In the cut rooted by O_1, only four vertices are replaceable since other vertices connect to the outside vertex O_2 or O_3. In the MFFW, the internal vertices are all replaceable.

Recently, Riener et al. [6] extended the cut structure to reconvergence driven windows and proposed a Boolean resynthesis to optimize the windows without database. As a reconvergence driven window is a complex multi-output structure, it is difficult to store its exact synthesis in the database. Riener et al. only apply a heuristic resynthesis to optimize it. In this paper, we propose a novel multi-output sub-structure, called fanout-free window (FFW), and its fully expansion maximum fanout-free window (MFFW). We also propose a new database dependent rewriting framework. Our main contributions are summarized as follows.

- We propose a novel multi-output sub-structure called fanout-free window. On the basis of FFW, we propose the fully window MFFW expanded with given inputs.
- We propose a database dependent rewriting framework, which supports both runtime cache and pre-computed database modes. Due to the large scale of different windows, we only consider the equivalent MFFWs from benchmarks. To avoid redundant calculations of equivalent windows, we only optimize a class of equivalent windows once and store the optimal one in the database. Furthermore, instead of enumerating all k-input windows, the framework enumerates MFFWs based on the leaves of k-input cuts.
- We propose the semi-canonicalization of windows to improve the efficiency of database. It takes high time complexity to check equivalence and classify windows by enumeration. The proposed semi-canonicalization fix the negation and permutation by invariant value of inputs and outputs.

*Xuliang Zhu and Ruofei Tang contribute equally to this work.
†Lei Chen is the corresponding author.



(a) An example of fanout-free window (b) Cut rewriting (c) MFFW rewriting

Fig. 2: An example of MFFW rewriting in DIV benchmark.

In doing so, we can calculate the truth value and classify windows in the database more efficiently.

- We conduct extensive experiments on EPFL benchmarks. The results show that semi-canonicalization reduces more than 98% fanout-free windows in the database and the proposed rewriting method outperforms the state-of-the-art methods nearly 50% in terms of node reductions.

II. Background and Related Works

A. And-Inverter Graph and Fanout-Free Windows

And-Inverter Graph. An And-Inverter Graph (AIG) is a data structure to model combinational logic circuits. Let $G = (V, E, \mathsf{PI}, \mathsf{PO})$ be an AIG, where (V, E) is a directed acyclic graph, PI is the set of primary input vertices, and PO is the set of primary output vertices. Each vertex $v \in V$ represents an And gate and $v \in \mathsf{PI}$ represents a primary input. Edges represent wires and can either be regular or complemented. For each vertex $v \in V$, $\mathsf{FI}(v)$ and $\mathsf{FO}(v)$ are the fanin and fanout vertices of v, i.e., in-neighbors and out-neighbors. For each vertex $v \in V$, $|\mathsf{FI}(v)| = 2$, and for each primary input vertex $v \in \mathsf{PI}$, $|\mathsf{FI}(v)| = 0$.

K-Input Fanout-Free Windows. Given a set I of k input vertices , a k-input fanout-free window (FFW) $G = (V, E, I, O)$ is also an AIG, where I are the input vertices and $O \subset V$ are the output vertices. A fanout-free window should satisfy following fanout-free rules,

1. For each vertex $v \in V$, $\mathsf{FI}(v) \subset V$.
2. For each vertex $v \in V \setminus O$, $\mathsf{FO}(v) \subset V$.

Specifically, for any vertex $v \in V \setminus O$, its fanins and fanouts are also in the fanout-free window. We call the vertices set fanout-free area [7]. Roughly speaking, all vertices in a fanout-free window must be able to be fully expressed in the form of Boolean function by vertices of PI. A k-input cut [7] is also a k-input fanout-free window, which is rooted by a given output vertex r. Note that the number of outputs in a cut may be larger than 1. Given k vertices I, the k-input maximum fanout-free window (MFFW) is fully expanded by inputs I, whose fanout-free area is maximum.

Vertex Boolean Function and Truth Table. Given a fanout-free window $G = (V, E, I, O)$ and a vertex $v \in V$, a vertex Boolean function is $f_v(x) : \mathcal{B}^k \to \mathcal{B}$, where $\mathcal{B} = \{0, 1\}$, $k = |I|$, and $x = (x_1, x_2, ..., x_k)$ is a list of Boolean variables of input vertices. A truth table $\mathsf{Truth}(v)$ can be represented as a 2^k-length bit string corresponding to 2^k input cases of f_v. Two vertices are complementary if their truth tables are complementary. The output truth tables of a window G is the set $T(G) = \{\mathsf{Truth}(o_1), ..., \mathsf{Truth}(o_{|O|})\}$, where $o_i \in O$. A 1-cofactor / 0-cofactor of $\mathsf{Truth}(v)$ regarding to input u is that 2^{k-1} input cases of f_v, where the Boolean variable of input u keeps 1 / 0.

TABLE I: Truth tables of vertices in the example window.

Id	Truth table	Id	Truth table
I_0	0101,0101,0101,0101	9	0001,0001,0001,0101
I_1	0011,0011,0011,0011	10	0010,0010,0010,1010
I_2	0000,1111,0000,1111	11	0100,0100,0100,0000
I_3	0000,0000,1111,1111	12	1001,1001,1001,0101
4	0000,0000,1111,0000	13	1111,0000,0000,1111
5	0000,0000,0000,1111	14	1111,0000,0000,0000
6	0000,1111,0000,0000	15	0000,1111,1111,0000
7	1100,1100,1100,0000	16	1000,1000,1000,0000
8	0101,0101,0101,0000	17	0110,0110,0110,1010

NPNP Equivalent. NPNP transformation of a given fanout-free window refers to Negation and Permutation of its inputs and Negation and Permutation of its outputs. Given two fanout-free windows G_1 and G_2, if there exists an NPNP transformation G_1' of G_1, such that truth tables of G_1' and G_2 are equivalent, i.e., $T(G_1') = T(G_2)$, we call that G_1 and G_2 are NPNP equivalent, denoted by $G_1 \equiv G_2$.

Example 1: Fig. 2(a) shows an example of a 4-input maximum fanout-free window, where $\{I_0, I_1, I_2, I_3\}$ are the input vertices, $\{12, 9, 13\}$ are the output vertices and other vertices are the fanout-free area. The area with triangular dashed is a 4-input cut rooted by vertex 12. It is still a fanout-free window with inputs $\{I_0, I_1, I_2, I_3\}$ and outputs $\{12, 7\}$. While it is not a maximum fanout-free window because vertices $\{4, 6, 9, 13\}$ can also be added into the window. Table I shows the truth table of the example fanout-free window. Each bit represents the value of vertex Boolean function with the given input, e.g. the 6-th bit of vertex 9 equal to 0 represents $f_9(1, 0, 1, 0) = 0$. The fanout-free window in Fig. 2(a) is NPNP equivalent to the fanout-free window in Fig. 2(c) because we can flip output vertices 12 and 13 and reorder outputs, where $\{\mathsf{Truth}(9), \mathsf{Truth}(\overline{12}), \mathsf{Truth}(\overline{13})\} = \{\mathsf{Truth}(9), \mathsf{Truth}(17), \mathsf{Truth}(15)\}$.

Algorithm 1 K-Input MFFW Rewriting Framework

Input: An AIG $G = (V, E, \mathsf{PI}, \mathsf{PO})$, an integer K, and a k-input FFW database D_k.
Output: A new AIG $G^* = (V^*, E^*, \mathsf{PI}, \mathsf{PO})$, and a new k-input FFW database D_k^*.
1: **for** each k-input cut $C = (r, L)$ **do**
2: **if** Leaves L have been visited **then**
3: **continue**;
4: Construct MFFW $G' = (V', E', L, O)$;
5: Compress the MFFW by functional reduction;
6: Canonicalize the MFFW by Algorithm 2;
7: **if** $|G^*| < |G'|$, where $G^* \in D_k$ and $G^* \equiv G'$ **then**
8: Replace G' by G^*;
9: **else**
10: Optimize the MFFW G';
11: Update G' into the database D_k.
12: **return** G, D_k;

B. Related Works

Rewriting is a greedy algorithm for optimizing the AIG area by transformation of local substructures. Traditional approaches use a single output structure, cuts, as the basic unit for replacement, and pre-compute optimal representations for each NPN class [1], [2]. When extending cut inputs from 4 to up to 12, the number of NPN classes grows exponentially, which prohibits any kind of pre-computation. Therefore, Yang et al. [3] proposed to build a runtime library to store enumerated cuts with golden design from the AIG. State-of-the-art rewriting techniques extend the scope of replacing. Introduction of exact synthesis enables the substitution of the whole cut [4], [5], by means of generating the exact Boolean network with the given specification and a set of functions [8], [9]. All these methods utilize a simple substructure, single output cut. In our work, we extend the single-output cut to multiple-output fanout-free window and propose a novel database dependent rewriting framework, which is more effective than cut rewriting.

Boolean function classification for AIGs (as well as fanout-free windows) is a vital problem in our work for matching and retrieving fanout-free windows in the database. There are several solutions on the fast computation of NPN classification [10], [11], [12], but all of them only can be applied on single-output Boolean function. Some methods are proposed for Non-exact NPNP Boolean matching [13], [14]. While these methods only try to achieve the large number of output equivalences of two fanout-free windows, which is not applicable for our database. In our work, we propose a semi-canonicalization that could check equivalence and retrieve fanout-free windows in the database efficiently.

III. Framework

Traditional cut rewriting [2] utilizes the pre-computed database. Different from cut rewriting, database-dependent MFFW rewriting meets three major challenges. First, the number of fanout-free windows in the AIG may be large. Given any k vertices, it could construct a k-input FFW based on these inputs. So, it takes $O(n^k)$ to enumerate all k-input MFFWs, where n is the size of AIG. Second, it is difficult to verify the equivalence of fanout-free windows. For a single-output k-input cut, it takes at most $2 \cdot 2^k \cdot k!$ to check whether two cuts are NPN-equivalent. While, for multiple-output fanout-free windows, it takes at most $2^k \cdot k! \cdot 2^{|O|} \cdot |O|!$, which is impossible to enumerate all cases of equivalence. Last, it is difficult to construct FFW databases. For k-input fanout-free windows, there exists 2^{2^k} different vertex truth tables. So, in theory, there exists $2^{2^{2^k}}$ different fanout-free windows, e.g., 2^{256} for 3-input fanout-free windows. It is difficult to store all these fanout-free windows and calculate their exact synthesis.

To tackle these three challenges, we propose a novel MFFW rewriting framework. First, we only enumerate MFFWs based on the leaves of k-input cuts instead of all k inputs, since others are weak connected sub-structures. It could guarantee the number of k-input MFFWs will not exceed the number of k-input cuts. Second, we propose a novel method to canonicalize the fanout-free windows. It can avoid the framework to enumerate all orders and flips of input and output vertices. Last, only few different fanout-free windows exist in real datasets. So, we only construct database based on the MFFWs in the real datasets. To further reduce the database size, on the basis of canonicalization, we could merge the equivalent fanout-free windows and store the optimal one in our database.

Rewriting Framework. Algorithm 1 shows the proposed MFFW rewriting framework. First, it enumerates all k-input cuts [7] and constructs the MFFWs based on each set of the leave nodes of these cuts (Lines 1–4). Then, it simulates the MFFW and compresses the functional equivalent vertices, i.e., two vertices that are equivalent or complementary (Line 5). The objective is to guarantee all vertices are functionally different. After compression, it canonicalizes the MFFW and calculates the key value (Line 6). On the basis of key value, it could find whether there exists a better equivalent fanout-free window in the database (Lines 7–11). If a better equivalent window exists, the framework replaces it. Otherwise, the framework optimizes it and updates the result into the database. Note that the framework is able to run without an external database. The input database could be empty and stored in the cache.

IV. Techniques

A. Fanout-Free Window Semi-Canonicalization

Given two fanout-free windows with different structures, it takes at most $2^k \cdot k! \cdot 2^{|O|} \cdot |O|!$ times of enumeration to check whether they are NPNP-equivalent. Thus, it is time-consuming to check and retrieve the equivalent fanout-free windows in the database. To retrieve efficiently, we propose a semi-canonical form of fanout-free windows. Instead of enumeration, we could canonicalize the window and retrieve the equivalent one based on the key set. The general idea is to fix the negation and permutation of inputs and outputs using invariant information. A similar method is proposed in [3],

Algorithm 2 K-Input FFW Semi-Canonicalization

Input: A fanout-free window $G = (V, E, I, O)$.
Output: A fanout-free window $G' = (V', E', I', O')$, an NPNP-transformation record R.
1: Let G' be a copy of G;
2: Simulate fanout-free window G';
3: **for** vertex $v \in O'$ **do**
4:     **if** $\mathsf{popcnt}(v) > 2^k - \mathsf{popcnt}(v)$ **then**
5:         Flip output vertex v;
6:         Record the flip of v to R;
7: Sort inputs $v \in I'$ by $\sum_{u\in O'} |\mathsf{popcnt}_v^1(u) - \mathsf{popcnt}_v^0(u)|$;
8: Record the input vertex order of I' relative to I;
9: **for** vertex $v \in I'$ **do**
10:     **if** $\sum_{u\in O'} \mathsf{popcnt}_v^0(u) > \sum_{u\in O'} \mathsf{popcnt}_v^1(u)$ **then**
11:         Flip input vertex v;
12:         Record the flip of v to R;
13: Re-simulate fanout-free window G';
14: Sort outputs $v \in O'$ by $\mathsf{Truth}(v)$;
15: Record the input vertex order of O' relative to O;
16: **return** G', R;

Algorithm 3 Key Calculation

Input: A semi-canonicalized fanout-free window $G = (V, E, I, O)$.
Output: a sorted truth table list Key.
1: $Key \leftarrow \emptyset$;
2: **for** vertex $v \in O$ **do**
3:     **if** $\mathsf{Truth}(v) > \mathsf{Not}(\mathsf{Truth}(v))$ **then**
4:         $Key \leftarrow Key \cup \{\mathsf{Not}(\mathsf{Truth}(v))\}$;
5:     **else**
6:         $Key \leftarrow Key \cup \{\mathsf{Truth}(v)\}$;
7: Sort the truth table list Key by the truth value;
8: **return** Key;

[11], while it is for single-output cut which could only be applied to the cut database. First, we define the popcount and get three lemmas as follows,

Definition 1 (popcount): Given an input vertex $v \in I$ and an output vertex $u \in O$, popcount is the number of 1 in the Truth (u), denoted by $\mathsf{popcnt}(u)$. And 1-popcount / 0-popcount is the number of 1 in the 1-cofactor / 0-cofactor of Truth (u) regarding to input v, denoted by $\mathsf{popcnt}_v^1(u)$ / $\mathsf{popcnt}_v^0(u)$.

Lemma 1: Given a fanout-free window G and an output vertex v, $\mathsf{popcnt}(v)$ is invariant regardless of the negation and permutation of inputs and other outputs.

Lemma 2: Given a fanout-free window G and an input vertex v, $\sum_{u\in O} |\mathsf{popcnt}_v^1(u) - \mathsf{popcnt}_v^0(u)|$ is invariant regardless of the negation and permutation of inputs and outputs.

Lemma 3: Given a fanout-free window G and an input vertex v, $\sum_{u\in O} \mathsf{popcnt}_v^1(u) - \mathsf{popcnt}_v^0(u)$ is invariant regardless of the permutation of outputs and negation of other inputs.

Algorithm 2 shows the semi-canonicalization of a given k-input fanout-free window. On the basis of Lemma 1 and Lemma 2, we could first flip outputs v by $\mathsf{popcnt}(v)$ (Lines 3-6) and sort inputs v by $\sum_{u\in O} |\mathsf{popcnt}_v^1(u) - \mathsf{popcnt}_v^0(u)|$ (Lines 7-8). Because these two values are invariant regardless of the negation and permutation of inputs and outputs. After fixing the negation of outputs and permutation of inputs, we could flip inputs v following Lemma 3 (Lines 9-12). Note that we could not flip inputs first because flip outputs will influence the value of $\sum_{u\in O} \mathsf{popcnt}_v^1(u) - \mathsf{popcnt}_v^0(u)$. Finally, we could sort the outputs by their truth value and get the semi-canonicalization of window (Lines 13-15). The key list is the sorted truth value of all outputs.

However, there exist some cases that NPNP-equivalent windows have different semi-canonical forms. For example, in Fig. 2, fanout-free window (a) and (c) are both semi-canonicalized. They are NPNP equivalent but they have different semi-canonical forms. Table I displays the truth tables of the vertices in Fig. 2 with a given input order I_0, I_1, I_2, I_3. Window (a) and (c) have complementary output vertices pairs (v_{12}, v_{17}) and (v_{13}, v_{15}). All of the above vertices happen to have $\mathsf{popcnt}(v) = 8$, so none of the output vertices will be flipped by the semi-canonicalization process. So even with the same permutation and negation of the inputs, windows (a) and (c) are divided into different classes due to the unequal key list. In fact, for 4-input fanout-free windows, there exist $\binom{16}{8} = 12,870$ of $2^{16} = 65,536$ vertices with $\mathsf{popcnt}(v) = 8$. For these cases, our semi-canonicalization is difficult to break them into equivalence classes exactly. So, we propose a new key list in Algorithm 3. For the outputs whose truth value larger than its complement, we store the complement instead of the truth value. It could identify the equivalence class for vertices with $\mathsf{popcnt}(v) = 8$ easily. In the example, the new key list of window (a) is $\{\overline{\mathsf{Truth}(13)}, \mathsf{Truth}(9), \overline{\mathsf{Truth}(12)}\}$ and the new key list of window (c) is $\{\mathsf{Truth}(15), \mathsf{Truth}(9), \mathsf{Truth}(17)\}$, which are equivalent. Note that the new key list is only used to verify and classify FFW in the database, and it would not change the semi-canonicalization.

B. Implementation Details in Framework

In this section, we introduce several implementation details in our rewriting framework (Algorithm 1).

MFFW Construction (Line 4). We construct the k-input MFFW based on the leaves of cut. Given a set of k input vertices I, we first add them to the window. After that, for each vertex $v \in V$, we traverse all its fanouts $u \in \mathsf{FO}(v)$ and check fanins of u. If both of its fanins are in the window, we add u to the window. After traversing fanouts of all vertices in the window iteratively, we connect the edges and construct the MFFW of the given input vertices I.

Fanout-Free Window Compression (Line 5). In our semi-canonicalization, all output vertices should be functionally different, in which, for any two vertices u and v, $f_u(x) \neq f_v(x)$ and $f_u(x) \neq \overline{f_v(x)}$. So, in our framework, we compress the fanout-free window by functional reduction before semi-canonicalization. Similar to FRAIG [15], we simulate the fanout-free windows and merge the functionally equivalent vertices following the topological order. After the compression, all vertices in the fanout-free window are functionally different.

TABLE II: Number of windows in benchmarks and database.

# Windows	Benchmark	After Semi-canon.	Compression
3-input	397,722	817	99.79%
4-input	931,072	7,948	99.15%
5-input	2,286,595	43,327	98.11%

Fanout-Free Window Replacement (Line 8). Similar to the cut replacement, our replacing algorithm adds the best window to circuit and removes useless vertices. First, it matches the negation and permutation of their inputs and outputs by the NPNP transformation. The transformation is recorded during semi-canonicalization. Then, it adds the replacement window to the network and connects its inputs and outputs. Finally, the dangling vertices in the replaced window are removed.

Fanout-Free Window Optimization (Line 10). In our rewriting framework, a fanout-free window is still an AIG, so logic synthesis and resynthesis methods are applicable to optimize our FFW database. Exact synthesis [8], [9] could find the optimal multi-outputs window by SAT-solver, while it takes exponential time complexity and low-efficient in lots of cases. So, similar to [5], we could limit the time of SAT-solver in our framework. Heuristic resubstitution [6] re-expresses vertex function using other existed vertices with cost-free inversions. Multi-outputs resubstitution is more efficient than exact synthesis. The traditional AIG resynthesis methods rewrite and refactor are also applicable to optimize our database. In the experiments, we use resyn2 [16] to optimize fanout-free windows, which is a combinational synthesis of rewrite, refactor, and balance.

V. Experiments

We implement our method in C++ using ABC library [16] and EPFL logic synthesis benchmark [17]. The correctness of all results has been verified using combinational equivalence checking (CEC) implemented in ABC [16].

EXP1: Quality Evaluation of Semi-Canonicalization. Table II shows the number of fanout-free windows in benchmark and our database. Benchmark column counts the number of windows iterated through in all EPFL benchmarks, and the second column counts the number of unique semi-canonical windows in our database. Compression is the reduction rate of original windows after semi-canonicalization. With the compression rate of total windows to be 99.79% / 99.15% / 98.11% for 3 / 4 / 5-input, the results indicate lots of windows are NPNP-equivalent in real-world datasets. It proves our semi-canonicalization algorithm is effective to compress the database. Even for 5-input windows, less than 50 thousands windows are contained in the database, which takes only 20 MB of memory.

EXP2: Quality Evaluation of FFW Database. In this experiment, we analyze the performance of MFFW rewriting with different caching strategies and demonstrate the importance of the FFW database. In our implementation of all methods, we only use resyn2 [16] to optimize the fanout-free windows in database. We compare three different implementations of our framework, the first method (MFFW without DB) only optimizes each window by resyn2 without database. The second implementation (MFFW with cache) uses an empty database and builds it runtime. The last method (MFFW with DB) uses a pre-computed database constructed from the benchmarks. The last three columns of Table IV show the performance of three implementations. For effectiveness, all methods have the similar performance because we use the same framework and fanout-free window optimization. The pre-computed database performs best in efficiency and both cached and pre-computed database could reduce the running time greatly up to 100X. It shows the efficiency of our database.

TABLE III: Comparison of combinational logic synthesis.

Methods	Average Improv.	Total Time
resyn2;	13.15%	59.00
resyn2; MFFW -4;	16.01%	68.08
MFFW -4; resyn2;	16.69%	64.17
resyn2*;	17.41%	95.72
resyn2+;	18.09%	107.91

EXP3: Comparison with the state-of-the-art Rewriting. In this section, we compare our approach with state-of-the-art methods respectively for AIG rewriting. Cut rewriting [2] is a traditional AIG rewriting method. It enumerates all cuts and selects the best replacement in the pre-computed database. Drw rewriting is the latest and best version of cut rewriting in ABC. So, we only report Drw in our experiments. Window rewriting [6] is a heuristic windows method without database. It extends cut into multiple-output windows and only replace the reconvergence driven windows. For fairness, we set the input size $k = 4$ of all cuts, and windows.

Table IV shows the efficiency and effectiveness of all methods. Size is the number of nodes in the AIG and time is measured in seconds. The average size improvement of Drw and Window are 8.83% and 9.13% while our method can reach 13.50% with pre-computed database. Our MFFW rewriting framework improves nearly 50% of reduction size on average. On the circuit Priority, our MFFW rewriting performs especially well with the optimization rate up to 48.98%, much better than 12.88% of Drw rewriting and 19.43% of Window rewriting. The cache based method also has a competitive performance. It can be applied in applications where a pre-computed database is not allowed.

For efficiency evaluation, our MFFW rewriting performs the worst, while the time still remains on the same order of magnitude as state-of-the-art approaches. Window rewriting performs best because it only enumerates the reconvergence driven windows instead of all windows. This technique could also be applied in our framework to improve the efficiency.

EXP4: Comparison of Combinational Logic Synthesis. In this experiment, we evaluate the quality of our MFFW rewriting in combinational logic synthesis. Resyn2 is a combinational logic synthesis in ABC. Resyn2$*$ method inserts three MFFW in the resyn2 sequence and resyn2$+$ inserts four MFFW. Table III shows the result. After inserting one and four MFFW, it increases 3% and 5% improvement, which indicates that our proposed method is also effective in the sequence optimization.

TABLE IV: Comparison of different 4-input rewriting methods.

Benchmark		State-of-the-art Methods						Our MFFW Rewriting Methods								
		Drw Rewriting [2]			Window Rewriting [6]			MFFW without DB			MFFW with Cache			MFFW with DB		
Name	Size	Time	Size	Improv.	Time	Size	Improv.	Time	Size	Improv.	Time	Size	Improv.	Time	Size	Improv.
Adder	1,020	0.01	1,020	0.00%	0.00	892	12.55%	1.03	893	12.45%	0.02	892	12.55%	0.03	892	12.55%
Bar	3,336	0.02	3,141	5.85%	0.01	3,141	5.85%	6.21	3,141	5.85%	0.11	3,141	5.85%	0.07	3,141	5.85%
Div	57,247	0.69	41,197	28.04%	0.14	41,267	27.91%	155.24	40,860	28.63%	1.66	40,793	28.74%	1.15	40,432	29.37%
Hyp	214,335	2.29	213,149	0.55%	0.61	210,402	1.83%	536.77	206,846	3.49%	4.9	206,801	3.52%	4.13	206,794	3.52%
Log2	32,060	0.33	29,761	7.17%	0.12	30,294	5.51%	86.59	29,520	7.92%	1.15	29,518	7.93%	0.78	29,494	8.00%
Max	2,865	0.02	2,862	0.10%	0.00	2,862	0.10%	5.00	2,862	0.10%	0.09	2,862	0.10%	0.06	2,862	0.10%
Multiplier	27,062	0.25	24,754	8.53%	0.09	25,706	5.01%	59.86	24,325	10.11%	0.67	24,323	10.12%	0.66	24,279	10.28%
Sin	5,416	0.06	5,191	4.15%	0.02	5,158	4.76%	14.00	5,094	5.95%	0.92	5,094	5.95%	0.15	5,092	5.98%
Sqrt	24,618	0.79	18,481	24.93%	0.10	18,719	23.96%	51.63	18,369	25.38%	0.76	18,369	25.38%	0.51	18,369	25.38%
Square	18,484	0.16	17,758	3.93%	0.06	17,502	5.31%	39.59	17,082	7.58%	0.68	17,081	7.59%	0.38	17,081	7.59%
Arbiter	11,839	0.09	11,839	0.00%	0.01	11,839	0.00%	36.53	11,839	0.00%	0.21	11,839	0.00%	0.21	11,839	0.00%
Cavlc	693	0.00	684	1.30%	0.00	693	0.00%	1.20	672	3.03%	0.49	672	3.03%	0.04	672	3.03%
Ctrl	174	0.00	122	29.89%	0.00	116	33.33%	0.15	97	44.25%	0.10	97	44.25%	0.03	97	44.25%
Dec	304	0.00	304	0.00%	0.00	304	0.00%	0.00	304	0.00%	0.03	304	0.00%	0.03	304	0.00%
I2c	1,342	0.01	1,280	4.62%	0.00	1,340	0.15%	1.94	1,281	4.55%	0.32	1,280	4.62%	0.04	1,280	4.62%
Int2float	260	0.00	222	14.62%	0.00	258	0.77%	0.43	217	16.54%	0.15	217	16.54%	0.03	217	16.54%
Mem_ctrl	46,836	0.34	46,291	1.16%	0.10	46,417	0.89%	76.37	46,156	1.45%	5.71	46,138	1.49%	0.58	46,137	1.49%
Priority	978	0.01	852	12.88%	0.00	788	19.43%	2.14	499	48.98%	0.06	499	48.98%	0.05	499	48.98%
Router	257	0.00	246	4.28%	0.00	244	5.06%	0.67	244	5.06%	0.07	244	5.06%	0.03	244	5.06%
Voter	13,758	0.14	10,384	24.52%	0.05	9,593	30.27%	25.25	8,749	36.41%	0.94	8,611	37.41%	0.27	8,611	37.41%
Average Improv.				8.83%			9.13%			13.38%			13.45%			13.50%

VI. Conclusion and Future Work

In this work, we study the area-oriented logic optimization of And-Inverter Graphs. To improve the traditional cut rewriting, we propose a new multi-output structure MFFW. The general idea is to traverse MFFW and optimize them by exact synthesis or heuristic methods. To improve the efficiency and effectiveness, we propose a database dependent framework that stores the optimal sub-structures. We also propose the semi-canonicalization to reduce the size of database. Extensive experiments validate the quality of our proposed methods.

This paper also opens up some interesting questions. One challenging direction is how to trade-off the level and area optimization. In our framework, we only store the optimal windows with minimal size, while the replacement may increase circuit level. Combinational strategy of optimizing windows is another direction. Exact synthesis is more effective while less efficient, so it is worth developing a combinational strategy of exact synthesis and heuristic methods.

Acknowledgement

This paper is supported by HK RGC Grant No. 12200021.

References

[1] N. Li and E. Dubrova, “Aig rewriting using 5-input cuts,” in *2011 IEEE 29th International Conference on Computer Design (ICCD)*, 2011, pp. 429–430.

[2] A. Mishchenko, S. Chatterjee, and R. Brayton, “Dag-aware aig rewriting: A fresh look at combinational logic synthesis,” in *2006 43rd ACM/IEEE Design Automation Conference*, 2006, pp. 532–535.

[3] W. Yang, L. Wang, and A. Mishchenko, “Lazy man’s logic synthesis,” in *2012 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2012, pp. 597–604.

[4] H. Riener, W. Haaswijk, A. Mishchenko, G. De Micheli, and M. Soeken, “On-the-fly and dag-aware: Rewriting boolean networks with exact synthesis,” in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 1649–1654.

[5] H. Riener, A. Mishchenko, and M. Soeken, “Exact dag-aware rewriting,” in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020, pp. 732–737.

[6] H. Riener, S.-Y. Lee, A. Mishchenko, and G. De Micheli, “Boolean rewriting strikes back: Reconvergence-driven windowing meets resynthesis,” in *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2022, pp. 395–402.

[7] J. Cong, C. Wu, and Y. Ding, “Cut ranking and pruning: Enabling a general and efficient fpga mapping solution,” in *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, 1999, pp. 29–35.

[8] M. Soeken, G. De Micheli, and A. Mishchenko, “Busy man’s synthesis: Combinational delay optimization with sat,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, 2017, pp. 830–835.

[9] W. Haaswijk, M. Soeken, A. Mishchenko, and G. De Micheli, “Sat-based exact synthesis: Encodings, topology families, and parallelism,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 4, pp. 871–884, 2019.

[10] X. Zhou, L. Wang, and A. Mishchenko, “Fast adjustable npn classification using generalized symmetries,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 12, no. 2, pp. 1–16, 2019.

[11] Z. Huang, L. Wang, Y. Nasikovskiy, and A. Mishchenko, “Fast boolean matching based on npn classification,” in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 310–313.

[12] X. Zhou, L. Wang, and A. Mishchenko, “Fast exact npn classification by co-designing canonical form and its computation algorithm,” *IEEE Transactions on Computers*, vol. 69, no. 9, pp. 1293–1307, 2020.

[13] C.-A. Wu, C.-J. Hsu, and K.-Y. Khoo, “Iccad-2016 cad contest in non-exact projective npnp boolean matching and benchmark suite,” in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2016, pp. 1–5.

[14] C.-W. Pui, P. Tu, H. Li, G. Chen, and E. F. Young, “A two-step search engine for large scale boolean matching under np3 equivalence,” in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2018, pp. 592–598.

[15] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, “Fraigs: A unifying representation for logic synthesis and verification,” ERL Technical Report, Tech. Rep., 2005.

[16] R. Brayton and A. Mishchenko, “Abc: An academic industrial-strength verification tool,” in *International Conference on Computer Aided Verification*, 2010, pp. 24–40.

[17] L. Amarú, P.-E. Gaillardon, and G. De Micheli, “The epfl combinational benchmark suite,” in *Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)*, 2015.