Maximum Fanout-Free Window Enumeration: Towards Multi-Output Sub-structure Synthesis

Ruofei Tang[1,*], Xuliang Zhu[3,*], Lei Chen[2,†], Xing Li[2], Xin Huang[1], Mingxuan Yuan[2], 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] Antai College of Economics and Management, Shanghai Jiao Tong University, Shanghai, China
{csrftang, xinhuang, xujl}@comp.hkbu.edu.hk, zhu.xl@sjtu.edu.cn, {lc.leichen, li.xing3, yuan.mingxuan}@huawei.com

***Abstract*—Peephole optimization is commonly used in And-Inverter Graphs (AIGs) optimization algorithms. The efficiency of these algorithms heavily relies on the enumeration process of sub-structures. One common sub-structure is the cut, known for its efficient enumeration method and single-output characteristic. However, an increasing number of optimization algorithms now target sub-structures that incorporate multiple outputs. In this paper, we explore Maximum Fanout-Free Windows (MFFWs), a novel sub-structure with a multi-output nature, as well as its practical applications and enumeration algorithms. To accommodate various algorithm execution processes, we propose two different enumeration styles: Dynamic and Static. The Dynamic approach provides flexibility in adapting to changes in the AIG structure, whereas the Static method ensures efficiency as long as the AIG structure remains unchanged during execution. We apply these methods to rewriting and technology mapping to improve their runtime performance. Experimental results on pure enumeration and practical scenarios show the scalability and efficiency of the proposed MFFW enumeration methods.**

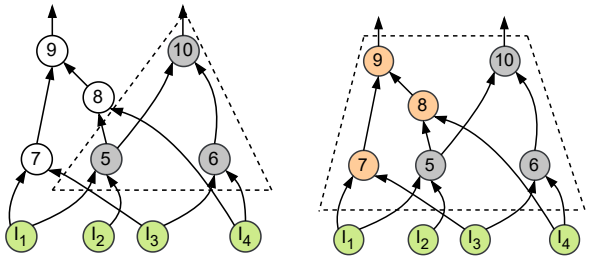
***Index Terms*—Logic Synthesis, Logic Optimization, And-Inverter Graph, Cut**

I. INTRODUCTION

And-Inverter Graph (AIG) is a robust representation of combinational logic circuits, consisting of two-input AND gates and NOT gates. Due to its simplicity, it has become widely used as a data structure for exploring technology-independent Logic Synthesis. Numerous studies have been conducted on AIG optimization aimed at reducing both size and delay [1], [2]. To ensure scalability, these algorithms often employ a peephole optimization approach, which focuses on achieving optimal implementation within sub-circuits rather than the entire circuit.

The traditional structure of sub-circuits in AIGs is known as cuts. A cut represents a cone-shaped region in the graph, as depicted in Figure 1(a), where a cluster of grey nodes illustrates a cut. Originally introduced by Cong as a representation for LUTs [3], cuts gained popularity in technology mapping methods. Early optimization algorithms focused on single-output sub-circuits and utilized fast enumeration techniques for cuts in AIGs [4], making them a suitable choice as sub-circuit structures. However, as more recent studies began to support the resynthesis of multiple-output sub-circuits, cuts revealed a limitation: size.

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



(a) Cut (b) 4-Input MFFW

Fig. 1. An example of a cut and 4-Input MFFW in AIG.

Several attempts have been made to extend the concept of cuts. For example, kl-cuts [5] expand cuts by introducing more outputs. However, kl-cuts suffer from inefficient enumeration and unstable structure scales, as will be discussed in Section III-A. Many optimization studies have explored cut expansion without introducing extra inputs [6]–[9]. The orange nodes in Figure 1(b) illustrate the expanded regions resulting from such cut expansions. However, this expansion process introduces additional runtime, significantly impacting the efficiency of structure enumeration.

Therefore, in this paper, we focus on a specific definition of expanded structures known as Maximum Fanout-Free Windows (MFFWs) [9]. We propose efficient enumeration methods to accelerate the enumeration process of synthesis algorithms. Our main contributions are listed as follows:

- We examine various workflows of logic synthesis algorithms and classify them into two frameworks: Dynamic and Static. The Dynamic framework modifies the structure during enumeration, while the Static framework preserves it throughout.
- For the Dynamic framework, we identify the current typical expansion process from cuts as Basic Enumeration. We analyze its limitations when dealing with large fanout nodes and propose a novel expansion method. Moreover, we propose a Dynamic Enumeration framework that automatically selects between two expansion methods.
- For the Static framework, our initial approach is a node-level arithmetic method. However, this method encounters performance bottlenecks when handling nodes with a high fanout. To overcome this challenge, we devise an enhanced approach, which merges cuts that share identical inputs.

978-3-9826741-0-0/DATE25/© 2025 EDAA

- We rigorously prove the correctness of our proposed methods. Furthermore, we analyze the time complexity of both Dynamic and Static enumeration methods.
- We apply the enumeration methods to MFFW rewriting to improve efficiency. To further demonstrate the potential of MFFW in technology mapping, we propose an MFFW-based analysis for priority-cut mapping [10] and use these methods to accelerate the process.
- We perform experiments on relatively large circuits to evaluate the efficiency of the proposed enumeration methods. The experimental results demonstrate that our proposed techniques can achieve a significant speed improvement of up to 100X.

II. PRELIMINARIES

A. And-Inverter Graph

The And-Inverter Graph (AIG) is a directed acyclic graph used to model structural implementation of Boolean functions [11]. An AIG G can be represented as a quadruple (V, E, PI, PO), where V is the set of logic And nodes, PI and PO are sets of primary input nodes and primary output nodes respectively, and $E \subseteq V^2 \cup (PI \times V) \cup (V \times PO)$. Given a node n, denote the fanin nodes set of n as $FI(n)$ and the fanout nodes set of n as $FO(n)$. For any $n \in V$, $|FI(n)| = 2$. For any $n \in PI$, $|FI(n)| = 0$. For any $n \in PO$, and $|FO(n)| = 0$. An edge in E can either be regular or complemented, and a complemented edge will invert the signal of its source node.

B. Cuts

A cut c of a node $r \in V$ in AIG G can be represented as a 2-tuple (r, L), where L is a set of nodes and $L \subseteq V$, such that each path starting from one node in PI to r passes through at least one node from L. For any $L' \subsetneqq L$, L' does not satisfy the above property. The L is called a leaf node set. The cut $c = (r, L)$ is called a k-input cut if $|L| = k$ and a k-feasible cut if $|L| \leq k$. Cuts can be computed in an algebraic way [4]. Given a node $n \in V$ in AIG G, let its input nodes be denoted as n_1 and n_2. The set operation $\bowtie$ is defined as follows:

$$A \bowtie B \equiv \{u \cup v | u \in A, v \in B, |u \cup v| \leq k\}$$

Let $\Phi(n)$ be the set of leaf nodes of all k-feasible cuts of n. Then we have:

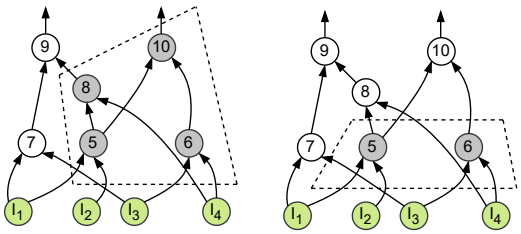
$$\Phi(n) = \begin{cases} \{\{n\}\} & : n \in PI \\ \{\{n\}\} \bigcup (\Phi(n_1) \bowtie \Phi(n_2)) & : \text{otherwise} \end{cases}$$

Using this formula, we can derive all k-feasible cuts for each node in the AIG in topological order [4].

C. K-Input Fanout-Free Windows

A fanout-free window (FFW) $W = (V', E', I, O)$ is a subgraph of an AIG $G = (V, E, PI, PO)$, where I is the window input nodes set that for any $n \in I, |FI(n)| = 0$. For any node $n \in V'$, $FI(n) \subseteq V' \cup I$ with $|FI(n)| = 2$. O is the set of window output nodes, such that for any $n \in O$, there exists $m \in FO(n)$, with $m \notin V$, and $O \subseteq V$.

The window input nodes set I should be the cut leaf node set for at least one node in the FFW. From a Boolean function perspective, for any $n \in V'$, the function of n can be fully expressed by only the elements in I. An FFW is k-input if $|I| = k$.



(a) 4-Input FFW (b) Incorrect FFW

Fig. 2. An example of an FFW and an incorrect example in AIG.

In accordance with its definition, an FFW $W = (V', E', I, O)$ possesses the following properties:

1. For each node $n \in V'$, there exists at least one node in I that is traversed by each path from a primary input node.
2. W forms a connected subgraph.

An FFW $W = (V', E', I, O)$ can be expanded by adding more nodes to V' and corresponding edges to E' from the AIG, once the new subgraph is still a valid FFW. If W cannot be expanded anymore, then the FFW is called a maximum fanout-free window (MFFW). In other words, V' of an MFFW contains all the nodes in the AIG whose functions can be fully expressed by the elements in I.

Example 1 Figure 1 and Figure 2 show the same one example of an AIG $G = (V, E, PI, PO)$, where $PI = \{I_1, I_2, I_3, I_4\}$, $V = \{n_5, n_6, n_7, n_8, n_9, n_{10}\}$ and PO nodes are omitted for simplicity. In Figure 1(a), the nodes in the dashed triangle with the leaf nodes I_1, I_2, I_3, I_4 form a 4-input cut, and n_{10} is the cut root node. In Figure 2(a), the area framed by the dashed line forms an FFW, with I_1, I_2, I_3, I_4 being input nodes. It is not maximum since n_7 and n_9 can be added to the FFW to form a larger valid FFW. Figure 1(b) presents an MFFW, which contains all the nodes that I_1, I_2, I_3, I_4 can express in function. Figure 2(b) shows an incorrect example of FFW, because the input set $\{I_1, I_2, I_3, I_4\}$ is neither a cut of n_5 nor n_6.

III. RELATED WORKS AND APPLICATIONS

In this section, we first review multiple-output structures from previous works and then introduce applications of MFFW and its enumeration methods.

A. Related Works

KL-Cuts. KL-Cuts are proposed as a multi-output structure [5], and have been applied to AIG optimization [12] and technology mapping [13]. The structure is based on two concepts: cuts and back-cuts, where back-cuts are cuts in the reverse AIG. It is formed by first enumerating an L-input back-cut, then combining K-input cuts of the back-cut leaf nodes. While useful, KL-Cuts have some drawbacks. First, the enumeration is inefficient. Despite efficient calculation of cuts and back-cuts, additional effort is required to check if the structure is closed and to exhaustively add nodes to complete it. Second,

the size of a KL-cut is unstable, ranging from L to K·L inputs, which leads to a large number of potential KL-Cuts. While this can be advantageous for some technology mapping methods, it is problematic for many optimization algorithms, which often require a fixed number of inputs, resulting in excessive caching and calculation.

Extended Cuts. This structure lacks a universal definition, but all variants extend from cuts, have a fixed number of inputs, and are multi-output, similar to FFW. The concept of *extended cuts* is introduced for exact synthesis-based rewriting [6] to explore a larger optimization space. In addition to expanding toward the PO direction, extended cuts also extend toward the PI direction without adding more inputs. Another related structure, called *window*, is used in rewriting [7]. Windows are enumerated with pruning based on reconvergence, but reconvergence is mainly useful for don't-care-based optimizations and is not applicable to all optimization algorithms.

These structures are enumerated inefficiently for using the **Basic Enumeration** in Section IV-A, allowing for potential improvements in efficiency.

B. Applications

AIG Optimization. For scalability, many optimizations replace local structures in the AIG. MFFW-like structures have been applied to expand the solution space, as mentioned earlier. Other works also utilize MFFW-like multi-output structures [8] [9], highlighting the need for more efficient structure enumeration. In this work, we introduce improved enumeration processes for MFFW rewriting, and experimental results demonstrate the effectiveness.

Technology Mapping. Many technology mapping methods analyze multi-output structures with a fixed number of inputs. For example, recognizing arithmetic sub-circuits, as in [14], uses cut enumeration and pattern recognition, merging related cuts into a 'forest' for analysis. Applying MFFWs and efficient enumeration can accelerate this process by quickly identifying outputs with shared inputs. Dual-output LUT mapping [15] also benefits from efficient enumeration.

In addition, we propose a new MFFW-based local analysis as a tie-breaker for If-map [10] to demonstrate MFFW's potential in technology mapping. Experiments testing its effectiveness are detailed in Section V-B

IV. Maximum Fanout-Free Window Enumeration

The process of expanding cuts to form an FFW is simple, but the challenge lies in ensuring it is maximal. Verifying maximality takes a computational cost of $O(n \cdot m)$, where n is the FFW size and m is the maximum fanout size, which is impractical for large FFWs. Therefore, it is essential to design the MFFW generation algorithms that theoretically guarantee the maximum size of the resulting FFW.

Another challenge is how different logic synthesis algorithms use MFFWs. We classify them into Dynamic and Static, as shown in Figure 3. Dynamic workflows modify each enumerated MFFW, making it difficult to cache cuts due to overlap between MFFWs, where changes to one can invalidate others.

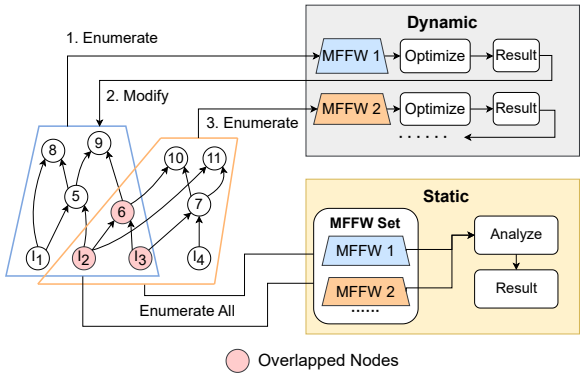


Fig. 3. Two styles of synthesis algorithm pipelines.

Algorithm 1 Regular Adding Nodes

Input: A node n in the FFW, a set of FFW nodes V.
Output: A set of nodes N that can be added to the FFW.

```
N = ∅
for each f_o ∈ FO(n) do
    if FI(f_o) ⊆ V then
        N = N ∪ {f_o}
return N;
```

Static workflows, which do not modify MFFWs, allow for more aggressive caching. To address both, we propose two enumeration methods tailored to these workflows.

A. Dynamic Enumeration

1) Basic Enumeration: We first introduce the basic method for enumerating MFFWs, which involves enumerating all cuts in the AIG and constructing MFFWs based on their leaf nodes. This process allows us to enumerate all MFFWs, as supported by the following lemma:

Lemma 1 *Given two MFFWs $W_1 = (V_1, E_1, I_1, O_1)$ and $W_2 = (V_2, E_2, I_2, O_2)$, $W_1 = W_2$ if and only if $I_1 = I_2$.*

Since the nodes functionally dependent on a given input set I are fixed and cover distinct regions for different I, enumerating cuts with different leaf nodes guarantees the enumeration of all MFFWs.

Once the theory is established, the next step is to construct windows from the input nodes. To add nodes to an FFW, we iterate through the fanouts of its current nodes and check if other fanin nodes belong to the FFW. Algorithm 1 provides the pseudocode for expanding an FFW by traversing the fanouts of a node. In the Basic Enumeration, this process repeats on all nodes in the FFW until no new nodes can be added.

However, the regular adding nodes process is extremely time-consuming when $|FO(n)|$ is large. For example, if a node with over 1,000 fanouts is encountered in the initial iteration round, then we have to traverse its fanouts many times, taking a lot of time. This is often the case in practice, where nodes with a large number of fanouts tend to appear in various MFFWs, thus making the process even more time-consuming.

2) Dynamic Enumeration: A common technique in AIG construction, *structural hashing (strashing)* [16], can help avoid traversing all fanouts for each node. Strashing assigns a unique number to each node and uses a hash table, with fanin node numbers and their polarities as the hash key.

Algorithm 2 Strashing Adding Nodes

Input: A node n in the FFW, a set of FFW nodes V.
Output: A set of nodes N that can be added to the FFW.

```
1: N = ∅;
2: for each node v ∈ V do
3:     N = N ∪ HashTableLookup(n, v)
       // HashTableLookup returns ∅ if no node is found.
4:     N = N ∪ HashTableLookup(n̄, v)
5:     N = N ∪ HashTableLookup(n, v̄)
6:     N = N ∪ HashTableLookup(n̄, v̄)
7: return N;
```

Algorithm 3 Dynamic K-Input MFFW Construction

Input: A set of leaf nodes L, an integer τ.
Output: An MFFW W.

```
1: Let S be a node-set
2: Let Q be a max-heap of nodes, sorted by fanout number
3: Insert nodes in L into Q
4: while Q ≠ ∅ do
5:     n = Q.pop_max()
6:     if |FO(n)| > τ · |S| then
7:         N = StrashAddNodes(n, S)
8:     else
9:         N = RegularAddNodes(n, S)
10:    Q = Q ∪ N
11:    S = S ∪ {n}
12:    Add nodes in N to W
13: Update corresponding edges in W
14: return W
```

Using strashing, we can efficiently add nodes to the window. As shown in the Algorithm 2, we can check whether node exists with fanins n and another node $v \in V$. If found, the node is added to the result set N. The function HashTableLookup returns the corresponding node, or an empty set if none exists.

Additionally, it is unnecessary to repeatedly apply the adding nodes algorithm to every node in the MFFW. Instead, performing the algorithm only once on each node is sufficient. If a node n, with $FI(n) = \{n_1, n_2\}$, and both n_1 and n_2 are already part of FFW, then applying adding nodes methods to either n_1 or n_2 will result in the addition of n to the FFW.

Incorporating the above improvements, we present the Dynamic K-input Window Construction in Algorithm 3. We use an integer τ (typically 4) to control whether to use strashing or regular adding methods. We use S to record the processed nodes and Q to store the nodes to process. Nodes with large fanouts are processed first since larger S takes more time to traverse, as nodes with smaller fanout nodes often contribute less to adding new nodes. This is a very practical strategy to reduce the running time.

No node in the MFFW with input nodes L is overlooked because all node pairs in W are checked for potential output additions. Any remaining nodes in the AIG have at least one fanin not in W.

After modifying the MFFW, the structural hash table must be updated. Since the table stores only local information (nodes with their two fanins), the update has linear time complexity.

B. Static Enumeration

Based on the comparable properties of the cut root and nodes in the window, we have the following lemmas:

Lemma 2 *Given is an AIG $G = (V, E, PI, PO)$ and one of its k-input FFW $W = (V', E', I, O)$. For each node $n \in V'$, there is at least one k-feasible cut $c = (n, L)$ that $L \subseteq I$.*

Proof Let m be a node in V'. Since $m \in V'$ so any path from primary input nodes to m should pass through at least one node in I. So there must be a minimal set $L' \subseteq I$ such that $c = (m, L)$ is a valid cut.

Lemma 2 suggests that it is possible to construct an FFW by identifying the roots of cuts whose leaf node sets are subsets of the window input nodes. Further, it can be proven that an FFW constructed in this manner must be a MFFW:

Lemma 3 *Given an AIG $G = (V, E, PI, PO)$ and one of its k-input MFFW $W = (V', E', I, O)$. If a node $n \in V$ and there is a cut $c = (n, L)$ where $L \subseteq I$, then $n \in V'$.*

Proof Let us assume that there is a node $m \in V$ such that $m \notin V'$ and a k-feasible cut $c = (m, L)$, where $L \subseteq I$. Consider a set P that consists of all the nodes on the path from each node in L to m, as well as m itself. Then, $P' = P \setminus V'$ is a non-empty set. If we add nodes in P' and corresponding edges to W, the resulting subgraph W is still a valid FFW, which is contradictory to the premise that W is a MFFW. Thus all nodes serving as cuts roots of subsets I are included in V'.

Using these two lemmas, we propose two static enumeration methods:

1) Static Enumeration 1: We first present a node-level arithmetic method to enumerate MFFWs, similar to cut enumeration. Algorithm 4 outlines the complete enumeration process. The algorithm uses two passes over the AIG nodes to construct all MFFWs. In the first pass (lines 1 to 7), we find all the cuts whose $|L| = k$ using $CalculateCuts(n, k)$, which returns all k-feasible cuts for the node n. Each input set is assigned a unique number, and nodes in the set are associated with this number (lines 6-7). In the second pass (lines 8-13), a hashmap W is initialized to group nodes within the same MFFW. MFFW memberships are then calculated by intersecting fanin memberships (lines 10-13).

However, this method struggles with large-fanout nodes. These nodes often belong to many MFFWs, making intersection calculations inefficient. Their frequent appearance as operands in these calculations further degrades performance, especially when there are many such nodes.

2) Static Enumeration 2: To reduce the cost associated with traversing fanout nodes, we make use of the structural properties of MFFWs. There is a similarity between cuts and maximum fanout-free windows (MFFWs). Specifically, the root of a cut r must have all its paths pass through the leaf nodes from primary inputs to r. Similarly, all nodes in an MFFW must have their paths pass through the window input nodes from any primary input.

From this observation, based on Lemma 2 and Lemma 3, we propose another static enumeration method. Algorithm 5 outlines the process:

1. We initialize a hash table H where the keys are cut leaf node sets, and the values are the corresponding root

Algorithm 4 Static K-Input MFFW Enumeration 1

Input: An integer k, an AIG $G = (V, E, PI, PO)$.
Output: A maximum fanout-free window set S.

```
1: Let H be a hashmap: node → set
2: for each node n ∈ V in topological order do
3:     Cut set C = CalculateCuts(n, k);
4:     for each cut c = (n, L) ∈ C where |L| = k do
5:         x = UniqueNumber(L)
6:         for each node l ∈ L do
7:             Add x to H[l]
8: Let W be a hashmap: integer → node-set
9: for each node n ∈ V in topological order do
10:    Let in_0 and in_1 be FI(n)
11:    H[n] = H[n] ∪ (H[in_0] ∩ H[in_1])
12:    for each integer x in H[n] do
13:        Add n to W[x]
14: S = values of W
15: Update corresponding edges for MFFWs nodes in S
16: return S
```

Algorithm 5 Static K-Input MFFW Enumeration 2

Input: An integer k, an AIG $G = (V, E, PI, PO)$.
Output: A maximum fanout-free window set S.

```
1: Let H be a hashmap: node-set → node-set
2: for each node n ∈ V in topological order do
3:     Cut set C = CalculateCuts(n, k);
4:     for each cut c = (n, L) ∈ C do
5:         H[L] = H[L] ∪ {n}
6: for each key I of H where |I| = k do
7:     Node set Q = ∅
8:     for each subset I' of I do
9:         Q = Q ∪ H[I']
10:    Let W be a MFFW where W = (Q, ∅, I, ∅)
11:    Update corresponding edges and outputs in W
12:    S = S ∪ {W}
13: return S
```

nodes. (line 1)

2. We traverse the AIG in topological order, computing all k-feasible cuts at each node. The leaf node sets and root nodes are stored in the hash table. (lines 2-5)
3. Using Lemmas 2 and 3, we construct MFFWs for valid window input sets. A valid window input set is a leaf node set of cardinality k. We add all root nodes of cuts with leaf nodes as a subset of I to $H[I]$, which now contains all nodes in the MFFW with I as input. Finally, edges are added, and the MFFW is inserted into the set S. (lines 6-12)

C. Time Complexity Analysis

We list the time complexity for each method in Table I. Here, S is the number of nodes in the AIG, N is the number of MFFWs, M is the sum of number of nodes in MFFWs, k is the size of MFFWs input, n in the number of nodes in a MFFW, and m is the maximum $|FO|$ of nodes in the AIG.

V. EXPERIMENTS

We implemented the MFFW enumeration methods and their applications in C++ using the ABC library [17]. The cut enumeration algorithm is supported by the ABC system. For our experiments, we selected large circuits from the IWLS'05 [18] and EPFL [19] benchmarks, each containing over 20,000 nodes.

TABLE I
TIME COMPLEXITY OF PROPOSED METHODS.

Method	Single MFFW	Total
Basic	$O(n \cdot m)$	$O(M \cdot m)$
Dynamic	$O(\min(n^2, n \cdot m))$	$O(M \cdot n)$
Static1	-	$O(S \cdot m^k)$
Static2	-	$O(M + N \cdot 2^k)$

TABLE II
STATISTICS OF 4-INPUT CUTS AND MFFWS IN BENCHMARKS

Benchmark	Size	Top 1% Avg	Cut #	MFFW #	Avg Size
div	57K	118.5	331,791	241,240	7.36
hyp	214K	255	2,296,564	1,068,140	8.53
log2	32K	119	532,736	214,598	8.23
multiplier	27K	65	451,890	155,924	8.74
b17	52K	394	338,744	213,481	7.81
b18	133K	426.2	927,406	572,704	7.97
b19	261K	422.09	1,762,939	1,111,852	7.91
des_perf	83K	33.25	612,489	441,818	8.91
leon2	792K	1463.59	3,032,787	2,962,037	6.34
netcard	864K	5522.9	3,336,235	2,849,339	6.52
vga_lcd	134K	2109.07	517,887	439,711	6.56
sixteen	16M	873.9	48,715,633	45,310,546	11.38
twenty	20M	835.71	62,466,409	59,048,160	11.14
twentythree	23M	886.1	70,287,092	66,550,042	11.02

We denote the running time as INF if execution exceeds 10 hours.

A. MFFW Enumeration

In this section, we focus on the characteristics of the benchmark circuits and the performance of MFFW enumeration.

EXP1: Statistics of the number of cuts and MFFWs in benchmarks. This experiment collects various circuit metrics that may impact the runtime of enumeration algorithms, as shown in Table II. Due to space constraints, only 4-input structure data is included, which will be used in later analysis. The definitions of the metrics are listed below:

- **Size**. The number of nodes in the AIG.
- **Top 1% Avg**. The average $|FO|$ for the top 1% of nodes by $|FO|$ value, indicating the distribution of large-fanout nodes.
- **Cut #**. The number of cuts with $|L| = k$.
- **MFFW #**. The number of MFFWs with $|I| = k$.
- **Avg Size**. The average number of nodes for MFFWs with $|I| = k$.

EXP2: Comparison of MFFW enumeration methods. Table III presents the runtime of each enumeration algorithm on benchmarks with MFFW input sizes of 4, 5, and 6. Basic Enumeration serves as the baseline since it is used in state-of-the-art optimization algorithms.

As k increases, the runtime of all methods grows. Static 1 and Static 2 exhibit similar growth rates because both are exponentially affected by k. The same applies to Basic and Dynamic methods, as both M and N in the time complexity analysis are also exponentially influenced by k, as shown in Table II.

In benchmarks with lower $Top1\%Avg$ values, all four algorithms perform comparably. However, in circuits with many large-fanout nodes and high extreme $|FO|$ values, performance differences emerge. For example, on leon2, netcard, and

TABLE III
COMPARISON OF MFFW ENUMERATION METHODS

Benchmarks	k=4				k=5				k=6			
	Basic time(s)	Dynamic time(s)	Static 1 time(s)	Static 2 time(s)	Basic time(s)	Dynamic time(s)	Static 1 time(s)	Static 2 time(s)	Basic time(s)	Dynamic time(s)	Static 1 time(s)	Static 2 time(s)
div	0.69	0.55	2.03	0.96	3.32	3.04	10.22	4.96	17.31	15.91	51.99	32.58
hyp	4.47	3.49	9.94	4.77	32.58	29.15	67.61	35.38	465.19	442.62	608.47	496.99
log2	1.10	0.92	2.16	0.94	17.83	15.59	19.46	12.90	83.94	79.28	108.49	82.82
multiplier	0.78	0.72	1.26	0.67	14.58	13.10	13.28	10.50	75.39	72.42	81.6	74.38
b17	1.60	1.40	2.56	0.90	14.05	14.06	12.66	4.73	43.50	45.70	63.42	27.72
b18	4.13	3.56	7.23	2.51	37.57	37.16	37.75	14.86	136.75	147.81	195.63	96.68
b19	8.52	7.01	14.19	5.08	75.72	72.70	72.26	27.77	262.13	261.49	363.14	178.40
des_perf	3.16	1.84	3.94	1.60	20.43	11.34	39.8	9.31	144.69	73.82	414.05	60.31
leon2	46.51	9.83	260.38	13.88	156.68	56.98	995.37	63.29	559.53	271.80	2770.35	304.31
netcard	524.92	9.17	3104.66	14.20	1,366.95	46.19	15428.8	57.88	INF	208.94	INF	281.80
vga_lcd	33.36	1.39	125.87	2.06	89.61	6.10	366.03	7.78	251.44	23.10	1002.41	31.23
sixteen	INF	2,691.97	INF	583.40	INF	16,469.18	INF	2,147.21	INF	INF	INF	6,959.78
twenty	INF	3,932.82	INF	786.37	INF	19,814.40	INF	2,571.32	INF	INF	INF	8,680.80
twentythree	INF	4,471.13	INF	1,056.62	INF	22,514.85	INF	2,867.43	INF	INF	INF	9,911.83

TABLE IV
4-INPUT MFFW REWRITING ON DIFFERENT ENUMERATION METHODS

Total	Basic [9]	Ours
Time(s)	165.61	75.90
Op Rate	20.24%	20.24%

TABLE V
RESULT OF MFFW-BASED 4-LUT MAPPING

Reduced LUT #	Basic(s)	Ours(s)	Called Times
2373	727.84	61.73	674K

vga_lcd, Basic and Static 1 show poor performance due to their heavy dependence on m. In very large circuits like sixteen, twenty, and twentythree, Dynamic Enumeration struggles when k exceeds 4, as the rapid increase in n becomes comparable to the average fanout. Static 2 proves to be the most adaptable, as it completely avoids enumerating fanout nodes.

Overall, our proposed approach significantly improves the enumeration speed compared to the baseline.

B. Applications of MFFW enumeration

In this part, we explore the application of MFFW enumeration algorithms. Due to space constraints, we present only the overall statistics of the experiments. The benchmarks sixteen, twenty, and twentythree from the *MtM* set are excluded, as their randomly generated nature makes their results unreliable for reference. We present our best performing method in these experiments.

EXP3: Application of Optimization. We implement MFFW rewriting based on [9] and apply our proposed enumeration methods to improve efficiency. Table IV shows the results for a single round of 4-input MFFW rewriting. The *Basic* refers to the original enumeration approach, while *Ours* represents our best-performing method. *Time* indicates the total runtime on the selected benchmarks, and *Op Rate* denotes the overall optimization rate, calculated as $1 - \frac{Optimized\_Node\#}{Original\_Node\#}$.

Compared to the baseline enumeration method from previous work, our proposed methods achieve significant runtime improvements without compromising the optimization rate. Notably, the total runtime for the Basic method is even lower than that of Basic Enumeration on the netcard benchmark in EXP2. This is due to the removal of many large fanout nodes during optimization, thereby avoiding the need to enumerate these large fanout nodes.

EXP4: Application of Technology Mapping. Based on If-Map [10], we propose a new cut sorting criterion, the *cover ratio*, to improve LUT mapping quality. Let V and O be the nodes set and output nodes set of an MFFW, then $cover\_ratio = \frac{|V|}{|O|}$. This metric estimates the average number of local nodes a cut can cover within the same MFFW. It works alongside *loose area flow*, which is similar to area flow but uses a much larger epsilon. Only if a significantly better cut is available will loose area flow make a decision; otherwise, the cover ratio is considered. We insert this mapping pass between the Area Flow and Exact Area passes and compare it with the ABC command *if -K 4*. Table V shows the experimental results. *Reduced LUT #* shows the number of LUTs reduced after applying our mapping pass. *Basic* and *Ours* refer to the total runtime on benchmarks, while *Called Times* reflects how many times loose area flow defers to the cover ratio for decision-making.

The results demonstrate the effectiveness of our mapping approach, with the *Called Times* highlighting the impact of our criteria in the workflow. Compared to the Basic enumeration method, our proposed methods significantly reduce runtime.

VI. CONCLUSIONS

In this paper, we introduce the problem of MFFW enumeration and propose both Dynamic and Static methods. Dynamic enumeration refines the basic approach, accommodating large fanout nodes and adapting to modifications in the AIG structure. Static enumeration, particularly Static 2, excels with the shortest average runtime by constructing MFFWs from cut leaf nodes. Our time complexity analysis is proven by experimental results, indicating the efficiency of our algorithms. We also prove the practicality of our methods through their application to various synthesis algorithms.

Our contributions set the stage for future window-based synthesis algorithms and raise some interesting questions. One notable question is adapting our methods to combinational circuit representations with multi-fanin nodes. In this situation, Addressing this scenario requires the development of alternative caching strategies beyond the Structural Hash technique.

REFERENCES

[1] A. M. R. Brayton, "Scalable logic synthesis using a simple circuit structure," in *Proc. IWLS*, vol. 6, 2006, pp. 15–22.

[2] W. Yang, L. Wang, and A. Mishchenko, "Lazy man's logic synthesis," in *Proceedings of the International Conference on Computer-Aided Design*, 2012, pp. 597–604.

[3] J. Cong and Y. Ding, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 1, pp. 1–12, 1994.

[4] 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.

[5] O. Martinello, F. S. Marques, R. P. Ribas, and A. I. Reis, "KL-cuts: a new approach for logic synthesis targeting multiple output blocks," in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE, 2010, pp. 777–782.

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

[7] 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)*. IEEE, 2022, pp. 395–402.

[8] F.-X. Reichl, F. Slivovsky, and S. Szeider, "Circuit minimization with QBF-Based exact synthesis," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 4, 2023, pp. 4087–4094.

[9] X. Zhu, R. Tang, L. Chen, X. Li, X. Huang, M. Yuan, W. Sheng, and J. Xu, "A Database Dependent Framework for K-Input Maximum Fanout-Free Window Rewriting," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[10] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts," in *2007 IEEE/ACM International Conference on Computer-Aided Design*. IEEE, 2007, pp. 354–361.

[11] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Proceedings of the 34th annual Design Automation Conference*, 1997, pp. 263–268.

[12] L. Machado and J. Cortadella, "Boolean decomposition for aig optimization," in *Proceedings of the on Great Lakes Symposium on VLSI 2017*, 2017, pp. 143–148.

[13] L. Machado, M. Martins, V. Callegaro, R. P. Ribas, and A. I. Reis, "KL-cut based digital circuit remapping," in *NORCHIP 2012*. IEEE, 2012, pp. 1–4.

[14] X. Wei, Y. Diao, T.-K. Lam, and Y.-L. Wu, "A universal macro block mapping scheme for arithmetic circuits," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 1629–1634.

[15] F. Wang, L. Zhu, J. Zhang, L. Li, Y. Zhang, and G. Luo, "Dual-output LUT merging during FPGA technology mapping," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.

[16] 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.

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

[18] C. Albrecht, "IWLS 2005 benchmarks," in *International Workshop for Logic Synthesis (IWLS): http://www. iwls. org*, 2005.

[19] 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)*, no. CONF, 2015.