

Ashenhurst-Curtis Decomposition Using Don't Cares

Benjamin Hien, Marcel Walter, Alessandro Tempia Calvino, Alan Mishchenko and Robert Wille

Abstract—Ashenhurst-Curtis decomposition (ACD) is a Boolean decomposition technique widely used in logic synthesis for tasks such as the decomposition of multi-valued relations, the encoding of multi-valued networks, and technology mapping into standard cells for ASICs and lookup tables (LUTs) for FPGAs.

A recent truth-table-based implementation of ACD has proven effective for delay-driven LUT mapping while also reducing the number of lookup tables for improved area efficiency. This method offers better runtime performance and a higher decomposition success rate, making ACD a practical and scalable technique for modern synthesis flows. However, it does not leverage the additional flexibility provided by don't-care conditions.

In this paper, we enhance ACD by incorporating controllability don't-cares extracted from cuts. By exploiting these additional degrees of freedom during decomposition, the proposed method achieves a higher decomposition success rate and a lower average number of LUTs per cut function. Specifically, we demonstrate that the decomposition success rate of practical functions into 6-LUTs increases from 51% to 53.4%, while the average number of LUTs per decomposition decreases from 2.50 to 2.46.

Moreover, in cases where state-of-the-art methods struggle to find valid decompositions—particularly with large fixed free sets—our method shows clear improvements. Success rates increase from 16.11% to 23.27% for four late-arriving variables and from 1.58% to 4.44% for five, with only a 1.5× runtime overhead.

I. INTRODUCTION

Ashenhurst-Curtis Decomposition (ACD) [1], [2], also known as Roth-Karp decomposition [3], is an effective Boolean decomposition algorithm that breaks down complex Boolean functions into smaller sub-functions and a composition function with reduced input support. By identifying a subset of variables and partitioning the function accordingly, ACD enables more efficient logic representations. This method has found wide application in logic optimization and technology mapping, including mapping to standard-cell libraries [4] and field-programmable gate arrays (FPGAs) [5], the decomposition of multi-valued relations [6], and the encoding of multi-valued networks [7].

In the context of FPGA design [8], ACD plays a key role in mapping Boolean functions into k -input lookup tables (k -LUTs) [5], which can implement any function with up to k inputs. This process, known as LUT mapping, is a central stage in FPGA synthesis flows. Modern LUT mappers operate on

a technology-independent graph representation of the circuit, referred to as the *subject graph*, which is first optimized independently of the mapping stage.

While this separation between optimization and technology mapping simplifies the overall flow, it also introduces challenges. The quality of the final mapping result is highly sensitive to the structure of the subject graph—a phenomenon known as *structural bias*. To improve mapping outcomes, synthesis flows employ *structural choices* [9]–[11], where multiple functionally equivalent representations are maintained to guide better mapping decisions. Additionally, techniques such as local collapsing and Boolean decomposition are applied to restructure logic for improved delay and area characteristics.

Among Boolean decomposition techniques, ACD is particularly effective. By partitioning the input variables into distinct subsets, it facilitates delay-driven technology mapping [12], wherein critical (i.e., late-arriving) variables are strategically assigned to the free set (FS). If a valid decomposition is identified under this configuration, it yields a two-level decomposition in which the FS variables serve as direct inputs to the composition function. As a result, these variables traverse only a single LUT, thereby minimizing their delay under the unified delay model. Recent advances in truth-table-based ACD [12] have demonstrated significantly faster runtimes and higher decomposition success rates compared to earlier methods [5], [6], [13], enabling improved mapping quality even for complex LUT architectures.

However, this ACD formulation does not exploit the optimization potential of don't-care conditions. Don't-cares represent input combinations for which the circuit output is irrelevant to its overall correctness and offer additional degrees of freedom that can be used to find valid decompositions.

In this work, we present a novel enhancement to ACD by incorporating don't-care information directly into the decomposition process. Our method leverages functional flexibility by utilizing *controllability don't-cares*, which are extracted from cuts identified during LUT mapping through windowing. This added flexibility allows the decomposition algorithm to achieve the following:

- 1) Enable more two-level decompositions for given sets of late-arriving variables, and
- 2) Produce decompositions that require fewer LUTs.

This suggests that, with our enhancement, delay-driven technology mapping and resynthesis engines can be made more powerful, provided an effective method for extracting don't-cares during mapping or resynthesis [14] is available.

Benjamin Hien, Marcel Walter, and Robert Wille are with the Chair for Design Automation, Technical University of Munich, Germany. Alessandro Tempia Calvino is with Synopsys Inc. and Alan Mishchenko is with UC Berkeley. Marcel Walter and Robert Wille are also with the Munich Quantum Software Company GmbH, Garching near Munich, Germany. Robert Wille is also with the Software Competence Center Hagenberg GmbH (SCCH), Austria. E-mail: {benjamin.hien, marcel.walter, robert.wille}@tum.de

x_2	x_1	x_0	f	cs	$\xrightarrow{x_0 \leftrightarrow x_2}$	x_0	x_1	x_2	f	cs
0	0	0	1	0	}	0	0	0	1	0
0	0	1	0	1		0	0	1	1	1
0	1	0	0	1	}	0	1	0	0	1
0	1	1	0	0		0	1	1	0	1
1	0	0	1	1	}	1	0	0	0	1
1	0	1	1	1		1	0	1	1	1
1	1	0	0	1	}	1	1	0	0	0
1	1	1	1	0		1	1	1	1	0
$f = 10110001$						$f = 0xA3$				

Fig. 1: Truth table representation with corresponding binary/hexadecimal encoding, cofactor extraction, and variable swapping of x_0 and x_2 .

The remainder of this paper is organized as follows. Section II introduces the necessary preliminaries, Section III reviews related work on state-of-the-art ACD, Section IV describes our don't-care-enhanced ACD approach, and Section V provides experimental results. Finally, Section VI concludes the paper and outlines directions for future work.

II. PRELIMINARIES

This section introduces the fundamental notations, definitions, and background related to logic networks, Boolean decomposition, and don't-cares.

A. Definitions

A *completely specified Boolean function* (CSF) is a mapping from an n -dimensional Boolean space to a single-dimensional one:

$$f : \{0, 1\}^n \rightarrow \{0, 1\}, \quad \text{where } n > 0. \quad (1)$$

A *truth table* representation of a k -input Boolean function $f : \{0, 1\}^k \rightarrow \{0, 1\}$ is a bit string $b = b_{l-1} \dots b_0$, i.e., a sequence of bits of length $l = 2^k$. A bit $b_i \in \{0, 1\}$ at position $0 \leq i < l$ corresponds to the function value under the input assignment $\vec{a} = (a_0, \dots, a_{k-1})$, computed as:

$$i = 2^{k-1} \cdot a_{k-1} + \dots + 2^0 \cdot a_0. \quad (2)$$

The *positive cofactor* of a Boolean function f with respect to a variable x_i , denoted as f_{x_i} , is obtained by setting $x_i = 1$. Similarly, the *negative cofactor*, denoted as $f_{\bar{x}_i}$, is derived by setting $x_i = 0$.

In truth table representations, the leftmost input variable (e.g., a_{k-1}) is commonly interpreted as the *most significant variable*, while the rightmost input variable (e.g., a_0) is the *least significant variable*. Swapping two input variables alters the structure of the truth table by exchanging the corresponding two-variable cofactors.

A *don't-care* condition in a Boolean function represents an input combination for which the output can be either 0 or 1. The set of all such input combinations forms the *don't-care set*. If a function has at least one don't-care condition,

it is referred to as an *incompletely specified Boolean function* (ISF). The complement of the don't-care set is the *care set*, which specifies input assignments where the output must be preserved. The care set can be represented by a truth table, where a value of 1 indicates a significant output and 0 indicates a don't-care condition.

Example 1. Consider the truth tables depicted in Figure 1. The functions are derived from one another by swapping the inputs x_0 and x_2 . For both truth tables, the cofactors with respect to the two most significant variables are shown. Additionally, their representations in binary and hexadecimal formats are provided.

An assignment of n Boolean variables to specific values is called a *minterm*. A *positive minterm* evaluates to 1, whereas a *negative minterm* evaluates to 0. In a CSF, every minterm is assigned either 0 or 1, forming the *care minterms*. In an ISF, some minterms may have a don't-care condition, referred to as *don't-care minterms*.

A CSF is *compatible* with an ISF (i.e., it implements the ISF) if it can be obtained by assigning either 0 or 1 to each don't-care minterm. One ISF is considered *larger* than another if it contains more don't-care minterms.

A Boolean function f *essentially depends* on a variable v if there exists at least one input assignment for which toggling v changes the function's output.

The *support* of f is the set of all variables on which f essentially depends. Two functions have *disjoint supports* if they share no common variables. A set of functions is *disjoint* if the supports of all functions in the set are pairwise disjoint.

A *Boolean network* is a directed acyclic graph (DAG), where each node represents a Boolean function. The graph's sources are the *primary inputs* (PIs), and its sinks are the *primary outputs* (POs). A node may have multiple *fanins* (nodes driving it) and *fanouts* (nodes it drives). If there is a path from node a to node b , then a belongs to the *transitive fanin* (TFI) of b , and b belongs to the *transitive fanout* (TFO) of a . The TFI of b , denoted $\text{TFI}(b)$, includes b and all nodes on paths from the PIs to b . Similarly, $\text{TFO}(b)$ includes b and all nodes on paths from b to the POs. If the nodes represent k -input lookup tables, the network is called a k -LUT network.

An *And-Inverter Graph* (AIG) is a Boolean network where all internal nodes are two-input AND gates. Inversions are not represented as separate nodes but are instead encoded as edge attributes.

A *cut* of a node n is a set of nodes, called *leaves*, such that:

- 1) Every path from any PI to n passes through at least one leaf node.
- 2) For each leaf node in the cut, there exists a path from some PI to n that passes through that leaf and no other leaf in the cut.

The node n is referred to as the *root* of the cut. A *trivial cut* contains only the node n and covers no other nodes. A *non-trivial cut* includes multiple nodes and covers all nodes on paths from the leaves to the root, excluding the leaves.

B. Boolean Decomposition

Boolean decomposition refers to the process of breaking down a Boolean function into smaller, simpler components. The result of such a decomposition is typically a Boolean network whose primary outputs (POs) remain functionally equivalent to the original function.

One of the most fundamental and well-known Boolean decompositions is the *Shannon decomposition*, which expresses a Boolean function f in terms of its cofactors with respect to a single variable x . It is defined as:

$$f = x \cdot f_x + \bar{x} \cdot f_{\bar{x}} \quad (3)$$

where $f_x = f_{x=1}$ and $f_{\bar{x}} = f_{x=0}$ are the positive and negative cofactors of f with respect to x .

The Shannon decomposition is a special case of more general Boolean function decompositions. One of the most fundamental and widely used forms is the *Ashenhurst-Curtis Decomposition (ACD)* [1]–[3], which provides a systematic way to decompose complex functions into simpler subfunctions by partitioning input variables into disjoint sets. The ACD of a single-output Boolean function f can be expressed as:

$$f(\vec{x}_{bs}, \vec{x}_{ss}, \vec{x}_{fs}) = g(\vec{h}(\vec{x}_{bs}, \vec{x}_{ss}), \vec{x}_{ss}, \vec{x}_{fs}) \quad (4)$$

Here, \vec{x}_{bs} , \vec{x}_{ss} , and \vec{x}_{fs} denote the *bound set (BS)*, *shared set (SS)*, and *free set (FS)*, respectively. These are disjoint subsets of variables that together form the support of f .

The function \vec{h} represents a vector of *BS functions*, which may be multi-output but typically have fewer outputs than the number of variables in the BS. The function g is the *composition function*, often implemented as a single k -input LUT in LUT-based decomposition.

The overall ACD structure is shown in Figure 2, where the BS, SS and FS all feed into the composition function.

C. Don't Cares

There can be two types of don't-cares in a logic network: *observability don't-cares* (ODCs) [15] and *controllability don't-cares* (CDCs) [16].

An ODC is associated with a node n and a primary input assignment $x \in \mathbb{B}^n$. The assignment x is considered *unobservable* at n if flipping the value of n does not affect any primary output. In such cases, x is an ODC of n because its value has no impact on the circuit's observable behavior. Although ODCs are effective for logic optimization, they require compatibility handling—such as computing *compatible ODCs* (CODCs) [15], [17]—to ensure correctness.

CDCs are computed by constructing a *window*, defined by collecting the TFI of a cut. The window is bounded by a leaf/root relationship: every path from a primary input to a root node must pass through at least one leaf, and the window includes all nodes on such paths, excluding the leaves themselves. To improve CDC quality, the window is often expanded to include *reconvergent paths*, where signals diverge from a common origin and reconverge through different fan-ins. The window is then simulated to enumerate all reachable

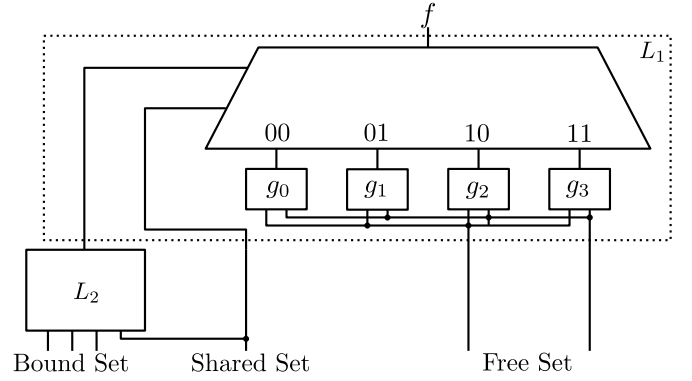


Fig. 2: General ACD structure with disjoint 4-variable *bound set (BS)*, 1-variable *shared set (SS)* and 2-variable *free set (FS)*.

input patterns at the cut leaves. CDCs are identified as those input combinations that never occur in the simulation. This procedure, known as *projection*, has exponential complexity in the number of cut leaves. This work focuses on CDCs, which can be safely used in synthesis and mapping without compatibility checks.

III. RELATED WORK

This section reviews related work on truth-table-based ACD [12], an efficient and scalable decomposition technique that enables delay-driven LUT mapping, resynthesis, and mapping into LUT cascade structures. The method structurally transforms Boolean functions using a multiplexer-like (MUX) decomposition pattern, as illustrated in Figure 2.

The approach begins by checking whether a valid decomposition exists through a partitioning of the function's support into three disjoint sets: the bound set (BS), shared set (SS), and free set (FS). As introduced earlier, placing late-arriving (i.e., critical) variables into the FS allows them to bypass additional logic levels, enabling delay optimization. For simplicity, only the BS and FS are considered in this section, leading to the following partitioning of the functional support:

- Let N be the number of variables in the support of the Boolean function to decompose.
- Let P be the number of variables selected for the FS; the remaining $N - P$ variables form the BS.
- The number of possible FS selections is $\binom{N}{P}$.
- To enable a feasible two-level decomposition into k -input LUTs, it is often useful to set $P = N - k$, so that the BS contains at most k variables.

Due to the MUX-like structure of Ashenhurst-Curtis decomposition, each assignment to the BS selects one cofactor (i.e., a P -input subfunction) defined over the FS variables. Therefore, for each FS partition, all such cofactors must be evaluated from the truth table:

- The truth table is first reordered with a dedicated procedure so that the FS variables occupy the least significant positions.

- With this ordering, the 2^{N-P} cofactors appear as contiguous blocks, each of size 2^P bits.
- These blocks are extracted at offsets $i \cdot 2^P$, with $i \in [0, 2^{N-P})$, directly from the reordered truth table.

Further, the number of *unique* FS functions is referred to as the *column multiplicity* μ . A decomposition into M BS functions is possible if:

$$\mu \leq 2^M \implies M \geq \lceil \log_2(\mu) \rceil. \quad (5)$$

If $P+M \leq k$, the composition function fits into one k -input LUT, and the decomposition is considered k -feasible.

To find such decompositions:

- All $\binom{N}{P}$ FS choices are iterated.
- For each FS, the number of unique FS functions (i.e., column multiplicity) is computed.
- If $M+P \leq k$ and $N-P \leq k$, the decomposition is immediately feasible.

Example 2. Consider the truth table on the left in Figure 1. Assume a FS of size 1, which results in FS functions of size $2^1 = 2$ bits each. Without considering don't-cares, all possible 2-bit FS functions appear—00, 01, 10, and 11—leading to the maximum column multiplicity of 4. When the variables x_0 and x_2 are swapped, the truth table changes accordingly, and so do the FS functions under the new variable ordering. The updated truth table, shown on the right, yields a reduced column multiplicity of 3. This demonstrates that by reordering variables and evaluating the resulting column multiplicity, the algorithm can determine whether a valid decomposition exists.

After evaluating that a decomposition exists, a functional encoding step, combined with support minimization, is used to derive the BS functions and the composition function, resulting in the final ACD structure. Since these steps are not modified in this work, the authors refer to [12] for further reading.

IV. DON'T-CARE-BASED ACD

This section presents the core contribution of this work by extending the truth-table-based ACD formulation introduced in [12] to incorporate don't-care conditions. As described in Section III, ACD first evaluates whether a valid decomposition exists by partitioning the support into a *free set* (FS) and a *bound set* (BS). If decomposition is possible, the algorithm then attempts to maximize the support and derive functional encodings for both the BS and the composition function. In the don't-care-aware version of ACD, only the decomposition evaluation step is modified, which is the primary focus of this section.

A. Decomposition Evaluation Using Don't-Cares

For decomposition evaluation, a valid composition exists if the column multiplicity μ is sufficiently small, i.e., $\mu \leq 2^M$, where M is the number of BS functions. In other words, the number of unique FS functions determines whether decomposition is possible. In the state-of-the-art approach, all FS functions are treated as *completely specified functions* (CSFs), where each minterm is assigned either 0 or 1. The column

multiplicity is then computed by counting the number of unique FS functions.

When incorporating don't-care conditions, some FS functions are instead represented as *incompletely specified functions* (ISFs), which include don't-care minterms. This makes evaluating the number of unique FS functions non-trivial, as one ISF can be collapsed onto another ISF or onto a CSF if their outputs differ only on don't-care minterms. Such collapsing can reduce the number of unique FS functions, thereby lowering the overall column multiplicity μ and enabling more decompositions.

Example 3. Consider again the truth table on the left in Figure 1, which results in a column multiplicity of 4 when using ACD without don't-cares. Now assume that a care set (CS) is provided alongside the output function. A value of 0 in the care set marks a don't-care condition, meaning the output for that input can be freely assigned.

With don't-cares taken into account, the FS functions become: 0-, -0, 11, and -0. Among these, only 11 is fully specified. The others contain one don't-care each and can potentially be treated as equivalent to other FS functions. Although none of these partial functions can be merged with 11 (since they all have a fixed 0 where 11 has 1), they can all be collapsed into the function 00, which is consistent with their defined bits. As a result, instead of four distinct FS functions, we now only need two: 11 and 00. This reduces the column multiplicity to 2.

However, determining which functions to collapse is not a straightforward task. The goal of this work is to leverage the additional information provided by the care set to minimize column multiplicity. This enhancement introduces new computational challenges in the decomposition evaluation step, which are examined in the following section.

B. Complexity

The decomposition evaluation problem corresponds to finding a minimum-size set of CSFs, \mathcal{C} , that covers a given set of FS functions, \mathcal{F} , where each $f_i \in \mathcal{F}$ is either a CSF or an ISF, and each ISF must be compatible with at least one $c \in \mathcal{C}$. This constitutes a domain-specific variant of the classical *set cover* and *minimum hitting set* problems, both of which are known to be NP-hard.

The core challenge arises from the exponential number of completions for each ISF. Given an FS function of size P , there are 2^P input minterms. If d of these are don't-care minterms, then the number of compatible CSFs for a single ISF is 2^d , since each don't-care minterm can independently take on a value of 0 or 1.

Furthermore, the total number of distinct FS functions (CSFs or ISFs) grows as 2^{2^P} . For ISFs, excluding the fully unspecified function (where all outputs are don't-cares), the number of ISFs with exactly d don't-cares is given by:

$$N_d = \binom{2^P}{d} \cdot 2^{2^P-d}. \quad (6)$$

Algorithm 1: Column Multiplicity Minimization

```
1: Input: Set of FS functions  $\mathcal{F}$  (CSFs and ISFs)
2: Output: Covering set  $\mathcal{C}'$ 
3:  $\mathcal{C}' \leftarrow$  all CSFs in  $\mathcal{F}$ 
4:  $\mathcal{U} \leftarrow$  all ISFs in  $\mathcal{F}$ 
5:  $P \leftarrow$  size of the FS
6: if  $P \leq 2$  then
7:    $\mathcal{C}' += \text{CoverWithBestCSFs}(\mathcal{U}, \mathcal{C}, \mathcal{C}') // (\text{Algorithm 2})$ 
8: else if  $3 \leq P \leq 5$  then
9:    $\mathcal{C}' += \text{AddUncoveredISF}(\mathcal{U}, \mathcal{C}, \mathcal{C}') // (\text{Algorithm 3})$ 
10: end if
11: Return:  $\mathcal{C}'$ 
```

Hence, the total number of ISFs is:

$$|\mathcal{F}| = \sum_{d=1}^{2^P-1} \binom{2^P}{d} \cdot 2^{2^P-d}. \quad (7)$$

Due to this double-exponential growth, exact algorithms become impractical even for moderate values of P . Consequently, practical solutions employ heuristics and approximation algorithms to reduce column multiplicity by merging ISFs into compatible CSFs while ensuring the decomposition's correctness [18]. Although these heuristic methods do not guarantee an optimal solution, they offer scalable and effective means of minimizing column multiplicity in practice, thereby making don't-care-aware decomposition feasible for larger support sizes.

C. Greedy Column Multiplicity Minimization

To efficiently approximate the solution to the column multiplicity minimization problem, we employ several greedy heuristics inspired by classical covering algorithms. The overall strategy is outlined in Algorithm 1. First, all completely specified FS functions are collected and directly added to the final covering set, as shown in Line 3. Starting from Line 6, the remaining ISFs are processed using different strategies depending on the size of the FS:

a) *Case $P = 1$:* For $P = 1$, we have $2^{2^1} = 4$ CSFs: 00, 01, 10, and 11. The number of ISFs is:

$$|\mathcal{F}| = \sum_{d=1}^1 \binom{2}{d} \cdot 2^{2-d} = \binom{2}{1} \cdot 2^1 = 2 \cdot 2 = 4.$$

These correspond to: 0-, 1-, -0, and -1. All four ISFs can be covered by two CSFs. For example, the pair 00 and 11 covers all ISFs and is therefore *complementary*. Alternatively, the pair 01 and 10 also suffices.

The general greedy algorithm described for $P = 2$ (see Algorithm 2) iteratively selects the CSF that covers the largest number of remaining ISFs and adds it to the covering set. This approach also applies to the case $P = 1$, but the problem simplifies significantly. Due to the small number of CSFs and ISFs, and the existence of complementary CSF pairs that jointly cover all ISFs, it is sufficient to exhaustively evaluate all CSF pairs and select one that fully covers the ISF set.

Algorithm 2: Covering with Best CSFs for $P \leq 2$

```
1: Input: Set of uncovered ISFs  $\mathcal{U}$ , set of all CSFs  $\mathcal{C}$ ,
   current cover  $\mathcal{C}'$ 
2: Output: Updated cover  $\mathcal{C}'$ 
3: for all  $c \in \mathcal{C}'$  do
4:    $\mathcal{U} \leftarrow \mathcal{U} \setminus \text{CoveredISFs}(c)$ 
5: end for
6: while  $\mathcal{U} \neq \emptyset$  do
7:    $c^* \leftarrow \arg \max_{c \in \mathcal{C} \setminus \mathcal{C}'} |\text{CoveredISFs}(c) \cap \mathcal{U}|$ 
8:    $\mathcal{C}' \leftarrow \mathcal{C}' \cup \{c^*\}$ 
9:    $\mathcal{U} \leftarrow \mathcal{U} \setminus \text{CoveredISFs}(c^*)$ 
10: end while
11: Return:  $\mathcal{C}'$ 
```

b) *Case $P \leq 2$:* There are $2^{2^2} = 16$ CSFs. The number of ISFs is:

$$\begin{aligned} |\mathcal{F}| &= \sum_{d=1}^3 \binom{4}{d} \cdot 2^{4-d} \\ &= \binom{4}{1} \cdot 8 + \binom{4}{2} \cdot 4 + \binom{4}{3} \cdot 2 \\ &= 4 \cdot 8 + 6 \cdot 4 + 4 \cdot 2 \\ &= 64. \end{aligned}$$

For this small FS size, the problem remains tractable. Each CSF can be represented as a 4-bit vector, and compatibility with ISFs can be efficiently encoded using bitmasks. Specifically, for each CSF, a bitmask of size 64 can be precomputed, where each bit indicates whether the CSF is compatible with a corresponding ISF. This enables constant-time compatibility checks during the algorithm.

The greedy strategy is outlined in Algorithm 2. CSFs that already appear in the input FS set are included in the covering set \mathcal{C}' . In Line 4, each such CSF is used to remove the ISFs it covers from the uncovered set \mathcal{U} . The main loop begins in Line 6, where the algorithm repeatedly selects the CSF that covers the largest number of remaining ISFs, as shown in Line 7. This CSF is then added to the cover in Line 8, and the set \mathcal{U} is updated accordingly in Line 9. The process continues until all ISFs are covered. This bitmask-based greedy approach yields high-quality coverings while remaining computationally efficient.

c) *Case $3 \leq P \leq 5$:* For larger free set sizes $P = 3, 4, 5$, the number of possible CSFs and ISFs grows rapidly, making the enumeration of all ISFs to solve a minimum hitting set problem computationally infeasible. Specifically, we observe 6,304 ISFs for $P = 3$, over 52 million for $P = 4$, and more than 1.3×10^{13} for $P = 5$.

However, for a truth table of size 2^N , the number of FS functions that can actually be extracted is bounded by 2^{N-P} . This significantly reduces the number of FS functions encountered during decomposition. Moreover, the number of unique CSFs derived from these FS functions is typically even smaller.

Algorithm 3: Covering Uncovered ISFs for $3 \leq P \leq 5$

```
1: Input: Set of extracted FS functions  $\mathcal{F}$ 
2: Output: Covering set  $\mathcal{C}'$ 
3:  $\mathcal{C}' \leftarrow$  all CSFs in  $\mathcal{F}$ 
4:  $\mathcal{U} \leftarrow$  all ISFs in  $\mathcal{F}$ 
5: for all  $f \in \mathcal{U}$  do
6:   if exists  $c \in \mathcal{C}'$  such that  $f$  is compatible with  $c$  then
7:     continue
8:   else
9:      $\mathcal{C}' \leftarrow \mathcal{C}' \cup \{f\}$ 
10:  end if
11: end for
12: Return:  $\mathcal{C}'$ 
```

To efficiently handle this case, we employ the incremental covering strategy outlined in Algorithm 3. Instead of solving a full minimum hitting set problem, the algorithm first collects all CSFs observed among the extracted FS functions, as shown in Line 3. The remaining FS functions are treated as ISFs in Line 4. For each ISF, the algorithm checks whether it is compatible with any CSF in the current cover, as described in Line 6. If so, the ISF is considered covered. Otherwise, it is added to the cover as a new CSF in Line 9.

This approach avoids pairwise ISF compatibility checks and provides a lightweight yet effective approximation in practice.

V. EXPERIMENTS

This section presents an experimental evaluation of the proposed don't-care-based ACD algorithm and compares it against the state-of-the-art ACD approach on practical functions from the EPFL benchmark suite [19]. To this end, the benchmarks are mined for cuts of sizes 7 to 11 and classified by NPN-equivalence. For each NPN class, one care set is tracked to ensure the overall number of functions remains manageable for this evaluation. The algorithms were implemented in C++17 within the open-source logic synthesis framework *Mockturtle*¹, and the experiments were conducted on a machine equipped with an AMD Ryzen 7 PRO 6850U processor and 32 GB of DDR5 RAM.

A. Comparison With Baseline ACD

To enable a fair comparison between the two ACD algorithms, we track several key metrics. All experiments use decomposition into 6-input LUTs, consistent with the baseline used in the state-of-the-art approach [12]. For each configuration, we record the decomposition success rate, which refers to the percentage of decompositions where all late-arriving variables are included in the FS. Additionally, we report the average number of LUTs produced after decomposition, as well as the average runtime per decomposition.

These measurements are collected for cuts of size 7 to 11 variables. For each cut size, we vary the number of late-arriving variables from 0 to 5 and exhaustively enumerate all possible combinations, averaging the results over all outputs.

Table I shows the results of the proposed approach using CDCs with a window size of 11, compared to the state-of-the-art ACD method. The size 11 is based on prior work, where it offers a good trade-off between the number of extracted don't-cares and runtime, and is commonly used in rewriting [20] and resynthesis [21].

As expected, the decomposition success rate decreases for both ACD methods as the number of late-arriving variables increases, since including all late variables in the FS becomes less likely. Similarly, larger function sizes lead to lower success rates due to increased number of possible FS functions.

Across all input configurations, our don't-care-based ACD consistently outperforms the baseline in terms of decomposition success rate. The largest improvements are observed for smaller functions, particularly for cuts of size 7. In this case, the average success rate improves from 72.03% to 77.37%, representing an absolute increase of more than 5 percentage points. The effect is even more pronounced when more late-arriving variables are present. For instance, for functions with seven variables and five late-arriving variables, our method achieves a success rate of 15.92% compared to just 6.06% in the baseline—an improvement of nearly 10 percentage points. In relative terms, this means that our method can decompose over two and a half times as many functions in this scenario.

In addition to improving the number of successful decompositions, our approach also yields structurally better results. The average number of LUTs per decomposition is consistently lower than that of the baseline; for example, the overall average LUT count is reduced from 2.50 to 2.46. This suggests that, when integrated into a synthesis or mapping engine, the don't-care-based ACD can contribute not only to improved delay but also to area savings. These benefits come at the cost of increased computational effort, with the average runtime per decomposition increasing from 2.07 to 3.48, representing a 1.5 \times overhead, which is considered a reasonable trade-off given the observed improvements and the additional search space introduced by don't-care handling.

B. Impact Of Don't Care Extraction Quality

The benefit of using don't-cares becomes less significant as the function size increases. For functions with 11 variables, the amount of available don't-care information is limited, resulting in only a minor improvement in decomposition success. This highlights the correlation between the quantity of available don't-cares and the potential for optimization.

Table II presents a comparison of the ACD approach using CDCs extracted through windowing with sizes 11, 14, and 16. Increasing the window size allows a larger portion of the logic cone to be captured, yielding more effective don't-cares. A steady improvement in the average decomposition success rate is observed as the window size increases: from 53.04% with a window size of 11 to 55.15% with size 14, and 56.20% with size 16. Similarly, the average number of LUTs per decomposition decreases from 2.46 to 2.43 and 2.42, respectively, indicating that the effectiveness of don't-care-based ACD depends on the quality of don't-care

¹Available at: <https://github.com/lsils/mockturtle>

Table I: Comparison of SoTA ACD and ACD DC 11 across varying late input counts and input sizes.

N Late	Metric	7 vars		8 vars		9 vars		10 vars		11 vars		Avg	
		SoTA	DC 11	SoTA	DC 11	SoTA	DC 11	SoTA	DC 11	SoTA	DC 11	SoTA	DC 11
0	DC Success	100.00%	100.00%	100.00%	100.00%	98.05%	98.06%	90.20%	90.62%	32.88%	32.88%	84.23%	84.31%
	Avg LUT	2.13	2.07	2.46	2.35	2.51	2.44	2.52	2.50	2.00	2.00	2.32	2.27
	Time(s)	0.12	0.21	0.99	2.34	5.19	12.26	23.40	32.79	7.51	11.44	7.44	11.81
1	DC Success	100.00%	100.00%	100.00%	100.00%	97.57%	97.62%	83.24%	84.08%	16.53%	16.53%	79.47%	79.65%
	Avg LUT	2.33	2.24	2.70	2.59	2.76	2.69	2.71	2.69	2.00	2.00	2.50	2.44
	Time(s)	0.09	0.21	0.57	1.32	2.57	5.74	8.50	13.66	3.23	5.25	2.99	5.24
2	DC Success	100.00%	100.00%	100.00%	100.00%	93.47%	93.98%	60.65%	62.85%	7.11%	7.11%	72.25%	72.79%
	Avg LUT	2.57	2.46	3.05	2.92	3.08	3.01	2.82	2.81	2.00	2.00	2.70	2.64
	Time(s)	0.05	0.11	0.26	0.55	0.96	1.99	2.93	5.34	1.48	2.54	1.14	2.11
3	DC Success	87.31%	92.64%	74.23%	79.77%	68.45%	72.16%	30.50%	33.60%	2.56%	2.56%	52.61%	56.15%
	Avg LUT	2.78	2.68	3.19	3.07	3.34	3.26	2.87	2.86	2.00	2.00	2.84	2.77
	Time(s)	0.03	0.05	0.11	0.22	0.33	0.65	0.88	1.67	0.83	1.56	0.44	0.83
4	DC Success	38.82%	55.64%	19.15%	30.03%	12.79%	18.50%	9.07%	11.48%	0.70%	0.70%	16.11%	23.27%
	Avg LUT	2.55	2.63	2.77	2.79	2.84	2.85	2.87	2.87	2.00	2.00	2.61	2.63
	Time(s)	0.02	0.03	0.05	0.09	0.14	0.25	0.36	0.69	0.60	1.19	0.23	0.45
5	DC Success	6.06%	15.92%	1.13%	4.18%	0.41%	1.36%	0.20%	0.54%	0.11%	0.21%	1.58%	4.44%
	Avg LUT	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	Time(s)	0.01	0.02	0.04	0.07	0.10	0.20	0.27	0.55	0.62	1.32	0.21	0.43
Avg	DC Success	72.03%	77.37%	65.75%	69.00%	61.79%	63.61%	45.64%	47.20%	9.98%	10.00%	51.04%	53.43%
	Avg LUT	2.39	2.35	2.70	2.62	2.76	2.71	2.63	2.62	2.00	2.00	2.50	2.46
	Time(s)	0.05	0.11	0.34	0.77	1.55	3.52	6.06	9.12	2.38	3.88	2.07	3.48

Table II: Average results across all benchmarks for SoTA ACD and DC-based variants.

N Late	Metric	SoTA ACD	ACD DC 11	ACD DC 14	ACD DC 16
0	DC Success	84.23%	84.31%	84.46%	84.60%
	Avg LUT	2.32	2.27	2.244	2.226
	Time(s)	7.44	11.81	11.044	10.696
1	DC Success	79.47%	79.65%	79.95%	80.23%
	Avg LUT	2.50	2.44	2.406	2.388
	Time(s)	2.99	5.24	4.984	4.864
2	DC Success	72.25%	72.79%	73.69%	74.33%
	Avg LUT	2.70	2.64	2.600	2.572
	Time(s)	1.14	2.11	2.032	1.986
3	DC Success	52.61%	56.15%	58.97%	60.61%
	Avg LUT	2.84	2.77	2.734	2.708
	Time(s)	0.44	0.83	1.010	0.802
4	DC Success	16.11%	23.27%	27.53%	29.83%
	Avg LUT	2.61	2.63	2.602	2.606
	Time(s)	0.23	0.45	0.434	0.434
5	DC Success	1.58%	4.44%	6.33%	7.63%
	Avg LUT	2.00	2.00	2.000	2.000
	Time(s)	0.21	0.43	0.422	0.414
Avg	DC Success	51.04%	53.43%	55.15%	56.20%
	LUTs	2.50	2.46	2.43	2.42
	Time(s)	2.07	3.48	3.32	3.20

extraction. However, the employed procedure for don't-care extraction does not scale well enough for integration into a full technology mapper, which is why the evaluation is limited to this experimental setup.

VI. CONCLUSION

This paper presents an enhanced version of Ashenhurst-Curtis decomposition (ACD) that incorporates controllability don't-cares, extending the capabilities of existing truth table-based approaches. While prior work achieves strong decomposition rates for small functions or limited numbers of late-arriving variables, the proposed method consistently

outperforms it—particularly in scenarios with larger fixed free sets. For cases with four or five late-arriving variables, the decomposition success rate more than doubles on average. The results highlight the importance of high-quality don't-care extraction, as the effectiveness of the approach scales with the amount of extracted flexibility. This includes, for example, the use of compatible observability don't-cares or other forms of don't-care computation that may be available in mapping or resynthesis engines. However, integrating scalable don't-care extraction directly into a technology mapping flow remains an open challenge and is left for future work. Although this

step can introduce additional computational effort, it enables significantly improved decomposition outcomes and opens new opportunities for delay-optimized LUT mapping. As such, the method offers a promising path toward more flexible and timing-aware synthesis flows.

REFERENCES

- [1] R. L. Ashenhurst, "The decomposition of switching functions," in *Proceedings of the International Symposium on the Theory of Switching*, 1957, pp. 74–116.
- [2] J. P. Curtis, *A New Approach to the Design of Switching Circuits*. New York, NY, USA: D. Van Nostrand, 1962.
- [3] J. P. Roth and R. M. Karp, "Minimization over boolean graphs," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 227–238, April 1962.
- [4] R. Francis, J. Rose, and K. Chung, "Chortle: A technology mapping program for lookup table-based field programmable gate arrays," in *Proceedings of the 27th ACM/IEEE Design Automation Conference (DAC)*, 1990, pp. 613–619.
- [5] C. Legl, B. Wurth, and K. Eckl, "Computing support-minimal subfunctions during functional decomposition," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 6, no. 3, pp. 354–363, September 1998.
- [6] M. Perkowski *et al.*, "Decomposition of multiple-valued relations," in *Proceedings of the International Symposium on Multiple Valued Logic*, 1997, pp. 13–18.
- [7] J.-H. Jiang, Y. Jiang, and R. K. Brayton, "An implicit method for multivalued network encoding," in *Proceedings of the International Workshop on Logic Synthesis (IWLS)*, 2001, pp. 127–131.
- [8] S. Ray, A. Mishchenko, N. Een, R. Brayton, S. Jang, and C. Chen, "Mapping into lut structures," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2012, pp. 1579–1584.
- [9] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, "Logic decomposition during technology mapping," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 8, pp. 813–834, August 1997.
- [10] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2005, pp. 519–526.
- [11] A. Mishchenko, R. Brayton, and S. Jang, "Global delay optimization using structural choices," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2010, pp. 181–184.
- [12] A. T. Calvino, G. De Micheli, A. Mishchenko, and R. Brayton, "Enhancing delay-driven lut mapping with boolean decomposition," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.
- [13] R. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, August 1986.
- [14] A. Costamagna, A. T. Calvino, A. Mishchenko, and G. De Micheli, "Area-oriented resubstitution for networks of look-up tables," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2025.
- [15] H. Savoj and R. K. Brayton, "The use of observability and external don't-cares for the simplification of multi-level networks," in *Proceedings of the 27th ACM/IEEE Design Automation Conference (DAC)*. Orlando, FL, USA: ACM, 1990, pp. 297–301.
- [16] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Dag-aware aig rewriting: A fresh look at combinational logic synthesis," in *Proceedings of the International Workshop on Logic and Synthesis (IWLS)*, Berkeley, CA, USA, 2006, available at https://people.eecs.berkeley.edu/~alanmi/publications/2006/iwls06_rewriting.pdf.
- [17] H. Savoj, "Don't cares in multi-level network optimization," Ph.D. Dissertation, University of California, Berkeley, May 1992.
- [18] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli, "Espresso heuristic logic minimizer," in *Proceedings of the 22nd ACM/IEEE Design Automation Conference*. IEEE, 1984, pp. 40–45.
- [19] L. Amarù, P.-E. Gaillardon, and G. D. Micheli, "The epfl combinational benchmark suite," in *Proceedings of the 24th International Workshop on Logic and Synthesis (IWLS)*, 2015, pp. 1–5.
- [20] A. T. Calvino and G. De Micheli, "Scalable logic rewriting using don't cares," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, pp. 1–6.
- [21] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang, "Scalable don't-care-based logic optimization and resynthesis," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 4, no. 4, pp. 1–23, 2011.