Enhancing Delay-driven LUT Mapping with Boolean Decomposition

Alessandro Tempia Calvino, *Graduate Student Member, IEEE*, Giovanni De Micheli, *Life Fellow, IEEE*
Alan Mishchenko, *Senior Member, IEEE* and Robert Brayton, *Life Fellow, IEEE*

Abstract—Ashenhurst-Curtis decomposition (ACD) is a decomposition technique used, in particular, to map combinational logic into *lookup tables* (LUTs) structures when synthesizing hardware designs. However, available implementations of ACD suffer from excessive complexity, search-space restrictions, and slow run time, which limit their applicability and scalability. This paper presents a novel fast and versatile technique of ACD suitable for delay optimization. We use this new formulation to compute two-level decompositions into a variable number of LUTs and enhance delay-driven LUT mapping by performing ACD on the fly. Compared to state-of-the-art technology mapping, experiments on heavily optimized benchmarks demonstrate an average delay improvement of 12.39%, and area reduction of 2.20% with affordable run time. Additionally, our method improves 4 of the best delay results in the EPFL synthesis competition without employing design-space exploration techniques. Moreover, we use the new formulation to compute exact decompositions into fixed LUT cascade structures of two LUTs, which have efficient implementations in the architecture of AMD FPGAs. Compared to the state-of-the-art method, this new formulation leads to an average reduction of 6.22% in delay, 3.82% in area, and 3.09% in edge count for better run time.

Index Terms—Logic synthesis, Boolean decomposition, technology mapping, FPGA

I. INTRODUCTION

FIELD-Programmable Gate Arrays (FPGAs) are integrated circuits with configurable logic blocks and programmable interconnects. Unlike application-specific integrated circuits (ASICs), which are designed for a specific application and have a fixed configuration, FPGAs can be programmed many times, which comes at the cost of lower power-performance-area (PPA). FPGAs are widely used for rapid prototyping, in low-volume applications, and for hardware acceleration of specific tasks.

Logic synthesis for hardware designs intended to run on FPGAs shares similarities with those for ASICs, but the target primitive is a k-input *look-up-table* (LUT), capable of implementing any Boolean function up to k inputs. Specifically, this paper focuses on mapping technology-independent combinational logic into networks composed of k-LUTs.

State-of-the-art technology mapping into LUTs is performed through local substitutions applied to an initial graph representation, called the *subject graph*. The drawback of this approach is that the technology-independent optimization step and the technology mapping step are separated. Consequently, the impact of optimization on the quality of the final LUT network is hard to predict before mapping. Delay-optimal mapping for a fixed subject graph is feasible in polynomial time [1]. Area-optimal mapping is NP-hard [2]. Specifically, the structure of the subject graph highly influences the mapping quality. This is known as *structural bias*. To mitigate structural bias, the known methods compute structural choices for the subject graph and use them during mapping [3], [4], or collapse and decompose parts of the graph during mapping [5]–[7]. However, exact area and delay optimization during LUT mapping remain NP-hard [8], [9]. In this work, we use Boolean decomposition to enhance delay-driven LUT mapping.

On another note, the performance of modern FPGAs is limited by programmable interconnect. Specifically, the interconnect delay can be up to 5 times the intrinsic delay of a LUT because wires are routed through multiple switch boxes and routing channels. One solution adopted by FPGA vendors is to supplement programmable interconnect with non-routable connections between LUTs, creating LUT structures such as LUT cascades [10]. However, existing placement algorithms struggle to effectively utilize these connections because this requires introducing LUT structures after LUT mapping. Alternatively, Boolean decomposition has emerged as an efficient way of generating LUT structures during mapping [11].

The Ashenhurst-Curtis decomposition (ACD) [12], [13], also known as Roth-Karp decomposition [14], is a powerful technique to decompose a Boolean function into a set of sub-functions and a composition function with reduced support. ACD finds applications in logic optimization and technology mapping. The traditional formulation of ACD breaks the input variables into two groups: the bound set (BS) and the free set (FS). Other approaches to ACD [15] allow for a shared set (SS) when some functions in terms of the BS variables are buffers. The larger the SS size, the fewer sub-functions are required. For instance, Figure 1 shows an ACD of a function with BS, FS, and SS, resulting in three 5-input LUTs. Conventional methods leverage *binary decision diagrams* (BDDs) [16] to perform ACD [15], [17], [18]. More recent approaches use truth tables for functions up to 11 or 16 inputs [11], [19].

This paper has two main contributions. First, we revisit the formulation of ACD with SS to enhance its computationally efficiency in LUT mappers and post-mapping resynthesis engines performing delay optimization. Our algorithm is truth-table-based and flexible in the number of FS, BS, and SS
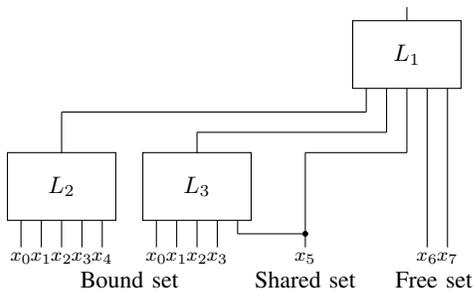
Fig. 1. ACD of an 8-input Boolean function into three 5-input LUTs with a 5-variable *bound set* (BS), a 1-variable *shared set* (SS), and a 2-variable *free set* (FS).

Fig. 2. Truth table representations and their encoding, cofactor extraction w.r.t. the two most significant variables, and variable swapping of x_0 with x_2.

variables, and in the number of BS functions. Our ACD runs up to 2x faster, compared to [11], and up to 80x faster, compared to [19], when performing decompositions into the LUT structure "66" composed of two 6-LUTs. Furthermore, the proposed method finds considerably more solutions, which translates into better quality. Second, we use ACD for the delay optimization of LUT networks. The idea is to compute functional decompositions using timing-critical variables in the FS and the rest of the variables in the BS and SS. This method is more general than cofactoring w.r.t. late arriving variables using Shannon expansion [20] and leads to improved quality of results. We integrate our ACD into the state-of-the-art LUT mapper for delay optimization. To our knowledge, this is the first practical and scalable work that uses ACD for delay-driven LUT mapping. Moreover, we utilize our new ACD approach to optimally map logic into LUT structures implementable using non-routable connections.

We experimentally evaluate the use of ACD for LUT mapping by comparing the results with state-of-the-art methods:

1) We show that the proposed ACD method has a higher decomposition success ratio, up to 32.58% more than state-of-the-art with a competitive run time.
2) We demonstrate that mapping with ACD can efficiently mitigate the structural bias and considerably reduce the delay. We compare the traditional LUT mapper in ABC, the LUT-structure mapper in ABC, and the proposed mapper with integrated ACD. We show that mapping with ACD notably outperforms the other mappers in delay by 7.52% on average, also when using structural choices [4]. Moreover, we show that an additional mapping round using the network obtained by ACD as a structural choice can further improve the delay, compared to the baseline, by 12.39%, with a surprising area reduction of 2.20%.
3) We present 4 new best results in the EPFL competition. These results have been obtained using delay-oriented mapping with ACD and without employing design-space exploration (DSE) methods. Hence, we expect even better results by using LUT mapping with ACD in a DSE tool.
4) We use this new ACD formulation to compute mappings into LUT structures composed of 2 LUTs with a non-routable connection between them. Compared to the state-of-the-art approach [11], our method reduces the average delay, area, and edge count by 6.22%, 3.82%, and 3.09%,

respectively, with better run time. In particular, this new formulation is exact, i.e., it always guarantees a solution for functions decomposable into 2 LUTs.

This paper is organized as follows. Section II provides the necessary background on logic networks, Boolean decomposition, and technology mapping. Section III introduces previous works on Boolean decomposition and LUT mapping. Section IV presents our ACD approach and its properties. Section V-B describes the integration of ACD into a LUT mapper for performance improvement. Section VI discusses the usage of our ACD formulation to leverage non-routable connections. Section VII presents experimental results and their interpretation. Finally, Section VIII concludes the paper.

II. PRELIMINARIES

This section introduces the basic notations and background related to logic networks, decomposition, and LUT mapping.

A. Definitions

A *Boolean function* is a mapping from a k-dimensional Boolean space into a 1-dimensional one: $\{0, 1\}^k \rightarrow \{0, 1\}$.

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

$$2^{k-1} \cdot a_{k-1} + \cdots + 2^0 \cdot a_0 = i. \tag{1}$$

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

It is common to refer to the leftmost input column of a truth table as the *most significant variable* (a_{k-1}) and the rightmost input column as the *least significant variable* (a_0). A *swap* of two variables alters the truth table by exchanging the location of the corresponding two-variable cofactors.

Figure 2 depicts two truth tables represented as bit strings, one in binary and one in hexadecimal. Notably, the rightmost truth table can be derived from the leftmost one by swapping variables x_0 and x_2. Marked next to both truth tables are the cofactors with respect to two most significant variables.

A completely specified Boolean function f *essentially depends* on a variable v if there exists an input combination,

such that the value of the function changes when the variable is toggled ($\frac{\partial f}{\partial v} = 1$). The *support* of f is the set of all variables on which function f essentially depends. The supports of two functions are *disjoint* if they do not have shared variables. A set of functions is disjoint if their supports are pair-wise disjoint.

A *binary decision diagram* (BDD) [16] is a logic representation based on the *if-then-else* operator. Each node in a BDD is associated with a variable and implements a cofactoring step. The root of a BDD is a node representing the given function, and the leaves are constant functions *true* and *false*. Each node is connected to two other nodes whose functions represent cofactors of the given function. The term BDD typically refers to *reduced ordered BDD* (ROBDD), which is a canonical representation for a given variable order and a set of reduction rules.

A *Boolean network* is modeled as a directed acyclic graph (DAG) having nodes associated with Boolean functions. The sources of the graph are the *primary inputs* (PIs), the sinks are the *primary outputs* (POs). For any node n, the *fanins* of n is a set of nodes driving n, i.e. nodes that have an outgoing edge towards n. Similarly, the *fanouts* of n is a set of nodes driven by node n, i.e., nodes that have an incoming edge from n. A k-*LUT network* is a Boolean network composed of k-input *lookup tables* (k-LUTs), capable of realizing any k-input Boolean function. An *and-inverter graph* (AIG) [21] is a Boolean network where nodes are 2-input ANDs and edges may implement inverters.

A *cut* C in a Boolean network is a pair $(n, \mathcal{K})$, where n is the node, called *root*, and $\mathcal{K}$ is a set of nodes, called *leaves*, such that 1) every path from any PI to node n passes through at least one leaf and 2) for each leaf $v \in \mathcal{K}$, there is at least one path from a PI to n going through v and not through another leaf. The *size* of a cut is the number of its leaves. A cut is k-feasible if its size does not exceed k.

B. Boolean decomposition

Boolean decomposition refers to the process of breaking down a Boolean function into simpler components. Boolean decomposition produces a Boolean network with POs functionally equivalent to the original function. The most generic decomposition is the Ashenhurst-Curtis decomposition (ACD) [12]–[14]. The ACD of a single-output Boolean function f can be expressed as follows:

$$f(\vec{x}_{bs}, \vec{x}_{ss}, \vec{x}_{fs}) = g(\vec{h}(\vec{x}_{bs}, \vec{x}_{ss}), \vec{x}_{ss}, \vec{x}_{fs}), \qquad (2)$$

where $\vec{x}_{bs}$ is the *bound set* (BS), $\vec{x}_{ss}$ is *shared set* (SS), and $\vec{x}_{fs}$ is the *free set* (FS). These sets are disjoint variable subsets, which together form the support of f. The function $\vec{h}$ may be multiple output with the number of outputs less than the BS size. The single-output functions in $\vec{h}$ are referred to as BS functions. The function g is referred to as the *composition function*. When decomposing into k-LUTs, the composition function is typically chosen to fit into one k-input LUT. Figure 1 shows an ACD of an 8-input function into three 5-input LUTs with a 5-variable BS, a 1-variable SS, and a 2-variable FS. The decomposition generates two BS functions (L_2, L_3) and a composition function (L_1).

The *disjoint-support decomposition* (DSD) [22] is a decomposition where the set of nodes have disjoint support. Hence, the Boolean network generated from DSD is always a tree. ACD generates a DSD decomposition when $\vec{x}_{ss} = \emptyset$ and BS functions have disjoint support.

The *Shannon decomposition* is a Boolean decomposition based on the Shannon expansion:

$$f = x f_x + \bar{x} f_{\bar{x}}. \qquad (3)$$

The result of applying the Shannon decomposition to all variables and merging identical cofactors, is a BDD.

C. FPGA technology mapping

LUT mapping is the process of expressing a Boolean network in terms of k-input lookup tables (k-LUTs). Before mapping, the network is represented as a k-*bounded* Boolean network called the *subject graph*, which contains nodes with a maximum fanin size of k. The AIG is the most common subject graph representation. The subject graph is transformed into a mapped network by applying local substitutions to sections of the circuit defined by cuts computed using cut enumeration [23]. A LUT mapper computes a mapping solution, called cover, by selecting a subset of the cuts that cover the subject graph while minimizing a cost function. State-of-the-art LUT mappers compute cuts and refine the cover in several mapping passes using heuristics based on delay, area, and edge count. For further details on LUT mapping, refer to [24].

III. RELATED WORK

A. Boolean decomposition

Traditionally, Boolean decomposition is implemented using BDDs [15], [17], [18], [25], derived by applying the Shannon decomposition to all variables in a given order and using reduction rules. Typically, multiple variable orderings are explored to find a partition of variables into *bound set* (BS) and *free set* (FS) and perform a support-reducing ACD [18]. However, algorithms that perform ACD suffer from slow run time and poor performance on large functions. To enhance efficiency, conventional methods often restrict decomposition to a limited set of primitives, such as 2-input operators and multiplexers [26], [27], and compute only *disjoint support decompositions* [22], [28].

Recent advancements have leveraged truth tables for ACD up to 16 variables, either by replicating variable re-ordering and size minimization of BDDs without explicitly constructing one or by computing a DSD that minimizes the required number of LUTs. Specifically, in [11], the authors use DSD and a heuristic variable re-ordering to find an ACD into a structure of 2 or 3 LUTs with non-routable connections. This method limits the shared set to at most one variable. In [19], the authors use ACD in post-mapping resynthesis when logic cones composed of several LUTs are collapsed into single-output Boolean functions and re-expressed using fewer LUTs by DSD and the Shannon expansion.

In this work, we address the limitations of previous ACDs. Our method is based on truth tables and does not have limitations on the number of LUTs and the size of SS. It

produces better quality of results and runs up to 2x faster than other more constrained implementations in ABC [11]. Moreover, our ACD does not rely on BDD heuristics or DSD with primitives but performs a more complete search.

B. LUT mapping

State-of-the-art LUT mapping for FPGAs relies on cut enumeration [23] followed by graph covering [1], [24]. Depth-optimal mapping for a k-bounded network is solvable in polynomial time [1], while area-optimal mapping is proven to be NP-hard [2]. The structure of the subject graph influences the structure of the mapped network to a large extent. This is known as *structural bias*. Mitigating structural bias is essential to improve the mapping quality.

Several methods derive an LUT network by applying flavors of Boolean decomposition to the BDD of the original func-tion [7], [18], [29]. Despite having a lower structural bias, these approaches are run-time intensive and limited to small functions, for which BDDs can be constructed. In practice, they rarely work for functions with more than 16 inputs.

To scalably reduce structural bias, previous work adopted different techniques. In [3], [4], structural bias is reduced by accumulating *structural choices* for the subject graph and using them during mapping. In [5], [6], [11], decomposition into k-LUTs is performed during technology mapping. In particular, the method in [11] integrates ACD into k-LUT map-ping to map logic into non-routable LUT structures composed of 2 or 3 LUTs. The approach extracts combinational logic cones with more than k inputs and decomposes them on the fly.

In this work, we perform on-the-fly decomposition similar to [11] but with two main differences. First, we utilize a more flexible and expressive ACD formulation. Second, our method can be customized for delay minimization.

IV. IMPROVEMENTS TO ACD

This section discusses a fast and versatile truth-table-based implementation of ACD with shared set for single-output functions. We propose several enhancements that make ACD readily applicable in LUT mappers and resynthesis methods. Figure 3 illustrates the ACD computation. The BS, SS, FS, and the number of BS functions used are flexible and determined during the decomposition. The composition function (L_1) is implemented as a multiplexer controlled by the outputs of the BS functions and the shared set. The FS functions, FS (g_i), drive the data inputs of the multiplexer. These functions become part of the composition function.

In this section, we first review the properties of the proposed ACD, showing that it is as generic as the original definition in [12]–[14] (Section IV-A). Then, we show how to efficiently check the existence of a feasible ACD and divide variables into three sets: FS, BS, and SS (Section IV-B). Next, we show how to compute the decomposition while minimizing the number of BS functions and their support (Section IV-C). Finally, we discuss an alternative method to maximize the number of variables in the shared set (Section IV-D).
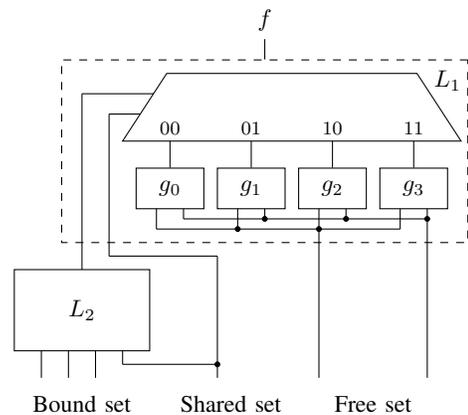

Fig. 3. Illustrating the AC decomposition of a Boolean function.

A. Properties of ACD

First, we formalize the definition of ACD and discuss its properties. Given the ACD shown in Figure 3 and the disjoint sets of variables $\vec{x}_{bs}, \vec{x}_{ss}, \vec{x}_{fs}$, we name

$$\vec{h}(\vec{x}_{bs}, \vec{x}_{ss}) = (h_0(\vec{x}_{bs}, \vec{x}_{ss}), \ldots, h_{v-1}(\vec{x}_{bs}, \vec{x}_{ss})) \quad (4)$$

the set of bound set functions of size $|\vec{h}| = v$. In Figure 3, $\vec{h}$ has size $v = 1$ and is represented by L_2. In Figure 1, $\vec{h}$ has size $v = 2$ and is represented by L_2 and L_3. An ACD can be expressed by Equation 2. In Figure 3, L_1 implements function g as a multiplexer with M select lines connected to functions in $\vec{h}$ and variables in $\vec{x}_{ss}$, such that $M = v + |\vec{x}_{ss}|$. An input assignment to the select lines of g selects a function $g_i(\vec{x}_{fs})$ where $0 \le i < 2^M$.

We demonstrate that our ACD decomposition is generic and includes other formulations, such as the Shannon decomposi-tion. Let us represent the function g as a ROBDD ordered with variables $\vec{h}$ and $\vec{x}_{ss}$ located close to the root, while variables $\vec{x}_{fs}$ are found close to the leaves. Let us draw a cut line in the ROBDD, such that nodes are partitioned into two disjoint sections: one dependent on $\vec{h} \cup \vec{x}_{ss}$ variables (denoted by α), and one dependent on $\vec{x}_{fs}$ variables (denoted by β). In our decomposition, α is implemented by the multiplexer of g, and β is implemented by the FS functions g_i. In particular, the number of nodes in β at the interface of the cut is equivalent to the number of unique g_i functions. Notably, we can extract β by drawing a cut in the ROBDD of f, with $\vec{x}_{fs}$ variables close to the leaves, separating $\vec{x}_{fs}$ from $\vec{x}_{bs} \cup \vec{x}_{ss}$ (see, e.g., [29], [30]).

Since in our representation of ACD function g implements a partitioned BDD, g is functionally complete, and ACD can implement any decomposable function. Moreover, the Shannon expansion (Equation 3) where x is a control input of the multiplexer, can be represented by ACD as follows:

$$f = f_x f_{\bar{x}} 1 + f_x \bar{f}_{\bar{x}} x + \bar{f}_x f_{\bar{x}} \bar{x} + \bar{f}_x \bar{f}_{\bar{x}} 0, \quad (5)$$

where x is a FS variable, f_x and $f_{\bar{x}}$ are BS functions, and FS functions g_i are 1, x, $\bar{x}$, and 0.

Definition 1: Variables in the SS that are not used by BS functions are called *independent shared set variables* (ISS

variables). Conversely, those that are used by BS functions are defined as *dependent shared set variables* (DSS variables).

According to the ACD definition in Equation 2, ISS variables would belong to the FS rather than the SS, since they are not in the support of functions in $\vec{h}$. However, in our decomposition, ISS variables serve as controls for a multiplexer, while the FS variables provide support for the FS functions, which feed into the data inputs for the multiplexer. We demonstrate that our definition is equivalent to the original one, i.e., if a decomposition with ISS variables in the SS exists, it also exists with ISS variables in the FS.

Theorem 1. *Let $\vec{x}_{ss} = \vec{x}_{iss} \cup \vec{x}_{dss}$ be an SS defined as the union of two disjoint sets: one of independent ($\vec{x}_{iss}$) and one not independent ($\vec{x}_{dss}$) SS variables. Then, $f(\vec{x}_{bs}, \vec{x}_{ss}, \vec{x}_{fs}) = g(\vec{h}(\vec{x}_{bs}, \vec{x}_{dss}), \vec{x}_{iss} \cup \vec{x}_{dss}, \vec{x}_{fs})$ can be written as $g'(\vec{h}(\vec{x}_{bs}, \vec{x}_{dss}), \vec{x}_{dss}, \vec{x}_{fs} \cup \vec{x}_{iss})$.*

Proof. Let us suppose that $\vec{x}_{iss}$ contains a single variable a. Function g is implemented as a multiplexer of M select lines connected to $\vec{h}$, $\vec{x}_{dss}$, and a, and 2^M data inputs functions $\{g_0, \cdots, g_{2^M-1}\}$. Then, each cofactor of g with respect to $\vec{h} \cup \vec{x}_{dss}$ variables is a function in the form $\dot{g}(a, \vec{x}_{fs}) = a \cdot g_i(\vec{x}_{fs}) + \bar{a} \cdot g_j(\vec{x}_{fs})$ with $0 \le i < j \le 2^M - 1$. Since the number of $\dot{g}$ cofactors cannot be larger than 2^{M-1}, f can be decomposed into the form $f = g'(\vec{h}(\vec{x}_{bs}, \vec{x}_{dss}), \vec{x}_{dss}, \vec{x}_{fs} \cup \{a\})$ with variable a in the free set. The generic case is proved by induction. ∎

Finally, we state a theorem used in Section V to conduct the search for a feasible decomposition.

Theorem 2. *If a decomposition of function f into 2 levels of k-LUTs with P variables in the free set does not exist, f cannot be decomposed with $P' > P$ variables in the free set.*

Proof. Let us suppose that a decomposition exists for $P' > P$ and does not exist for P. The decomposition with P' involves at most $k - P' + 1 < k - P + 1$ LUTs. This is a contradiction of the principles of information theory since a decomposition using P' has less information encoding than the one using P. ∎

B. Finding a feasible decomposition

After defining the properties of ACD, in this section we present an efficient method to check the existence of a Boolean decomposition and find an assignment of support variables to the FS and the BS (and SS). In particular, we focus on decomposition into a two-level k-input LUT structure. For simplicity, in this section, we include the SS variables in the BS.

The first step to derive a decomposition is to partition variables into FS and BS. Given a truth table, our approach enumerates different free sets. Let N be the number of variables in the support of the function to decompose. Let P be the number of variables to consider in the FS. The remaining $N - P$ variables are considered in the BS. The number of different free sets is $\binom{N}{P}$. Regarding the choice of value P when searching for a feasible two-level decomposition, for an N-input function and k-input LUTs, it is convenient to

consider $(N-k)$ variables in the FS, so that at most k variables are considered in the BS. For instance, when $N = 8$ and $k = 6$, we can choose $P = 2$ and evaluate $8 \cdot 7/2 = 28$ different 2-variable free sets.

For each FS, the truth table is transformed to have the FS variables as the least significant ones. The variable reordering is performed using a dedicated procedure, which swaps two variables at a time. Note that when enumerating all the free sets, the first FS composed of the P least significant variables in the support of the function does not need variable swapping, since the original truth table already reflects this order. Then, every consecutive FS can be derived from a previous FS by swapping one variable in $\vec{x}_{fs}$ with one in $\vec{x}_{bs}$. The complexity to explore all the FS is of $\binom{N}{P}$ swap operations. Figure 2 shows how a variable swap affects the truth table.

Each input assignment to the BS variables selects one P-input function in terms of the FS variables. Specifically, each P-input function is a cofactor with respect to variables in $\vec{x}_{bs}$. Given a truth table with this variable ordering, FS functions are easily computed by extracting groups of 2^P bits at $i \cdot 2^P$ offsets with $i \in [0, 2^{(N-P)})$. Informally, FS functions are bit-strings positioned next to each other in the bit-string of the truth tables. Figure 2 graphically depicts the extraction of cofactors with respect to the two most significant variables.

Example 1: Consider a 6-variable function represented in hexadecimal as the truth table $f = $ 0x8804800184148111. Assume that the FS variables are the two least significant variables and the BS variables are the four most significant ones. The functions in terms of FS variables have truth tables with $2^P = 4$ bits. There are $2^{(N-P)} = 16$ of these functions, corresponding to hexadecimal digits in the truth table (0x8, 0x8, 0x0, 0x4, etc). △

The target function can be realized using M bound set functions if the number of unique FS functions, known as column multiplicity μ, does not exceed 2^M, hence $M \ge \lceil \log_2(\mu) \rceil$. If $P + M \le k$, the composition function fits into one k-LUT.

Example 2: Continuing Example 1, there are 16 FS functions, of which only 4 are unique. The FS functions are 0x8, 0x0, 0x4, and 0x1. Hence, the column multiplicity $\mu = 4$, which requires $M = \lceil \log_2(4) \rceil = 2$ or more BS functions. Hence, this partition of variables into FS and BS produces a feasible support-reducing decomposition into 4-input LUTs. Using Figure 3, ACD assigns FS functions to g_i. Then, two BS functions of at most 4 inputs are necessary to select the correct FS function. △

We employ the enumeration of free sets while counting the number of unique cofactors to check if a support-reducing decomposition exists. In practice, a sufficient condition for a 2-level decomposition to exist, is to have $M + P \le k$ and $N - P \le k$, i.e., the composition function is k-feasible, and the number of remaining variables in the BS does not exceed k. However, a decomposition could have $N - P > k$ and k-feasible BS functions, as shown in Figure 1. In this case, it is not sufficient to partition variables into FS and BS to guarantee a 2-level decomposition (unless there are ISS variables that can be moved to the FS, by Theorem 1, to make $N - P \le k$ true). Consequently, each potential decomposition with $N - P > k$ and $P + M \le k$, similar to the one in Figure 1, must be

checked to be 2-level decomposable by computing minimal-support BS functions as shown in Section IV-C. Due to this additional computation, the latter ACD is often too slow to be used in mainstream LUT mappers or resynthesis engines.

After partitioning variables into FS and BS and computing the corresponding unique FS functions, our method uses the techniques in Section IV-C to produce a decomposition while minimizing the number of BS functions and their support.

C. Functional encoding and support minimization

Once a partition of variables into FS and BS with a feasible decomposition is found, the BS functions are extracted by assigning each FS function a code. Informally, an encoding represents the assignment of FS functions to the data inputs of the MUX of Figure 3 (e.g., the encoding of g_1 is 01). While any encoding that distinguishes FS functions is a valid solution, a good encoding also minimizes the number of BS functions and their support. It is crucial to find an encoding that minimizes the support for three reasons. First, if $N - P > k$, by minimizing the support, each BS function may ideally fit into a k-LUT, allowing for a two-level decomposition. Second, minimizing the support maximizes the shared set (an SS variable is a BS function represented by a buffer), reducing the number of required LUTs. Third, the number of edges is reduced, helping routability. Finding a feasible encoding is similar to solving constrained encoding problems [31]–[33].

An encoding assigns a code $T = t_{M-1} \ldots t_0$ of length M to each of the FS function. A variable t_i takes value 1, 0, or $-$, indicating the ON-set, OFF-set, and DC-set, respectively. A *minimum-length encoding* is an encoding of length $M = \lceil \log_2(\mu) \rceil$. An encoding is *strict* if a unique term T is assigned to each FS function, resulting in a Boolean function. An encoding is *non-strict* if multiple pair-wise disjoint terms T can be assigned to each FS function, resulting in a Boolean relation. For instance, given $M = 2$ and $\mu = 3$, an assignment "$1-$" to a FS function is strict, while "$01 \vee 10$" is non-strict. In this work, we only utilize strict encodings since non-strict encodings are too many to be efficiently enumerated in a fast ACD implementation. Moreover, experimental evaluations on practical functions suggest that non-strict encodings do not improve much the quality of the decomposition. For further details on the number of possible encodings, refer to [30].

Let *i-sets* be the set of μ Boolean functions in terms of the BS variables encoding FS functions using one-hot encoding. Specifically, an i-set represents one FS function and takes value 1 when an input assignment to the BS variables selects the corresponding FS function.

Example 3: Using Example 2, the i-set corresponding to the FS function 0x8 is 1100100010001000 in binary format. Note that the truth table depends on $N - P$ variables and has value 1 when the original function is 0x8. △

I-sets are used to derive a more compact encoding with a two-step procedure. The first step enumerates *candidate BS functions*. The second one solves a unate covering problem, in which columns are candidate BS functions and rows are pairs of FS functions to be distinguished.

Candidate BS functions are functions depending on BS variables whose output can used as t_i to encode FS functions.

They are enumerated by combining i-sets. To leverage all the functional degrees of freedom of a strict encoding, i-sets in a BS candidate can be either in the *ON-set*, *OFF-set*, or *don't-care* (DC) set. Since candidate BS functions are used as control inputs of a multiplexer, they can distinguish elements in the ON-set (takes value 1) against elements in the OFF-set (takes value 0). In encoding problems, BS functions are called *dichotomies*, while pairs of functions to be distinguished can be interpreted as *seed dichotomies* [33]. Don't-cares are also important to minimize the support, which translates into fewer LUT fanins.

Example 4: Continuing Example 3, let us consider the candidate bound set function h that has the i-sets {0x8, 0x1} in the ON-set, the i-set {0x4} in the OFF-set, and the i-set {0x0} in the DC-set. Its function in the binary format is $h =$11-01--110101111 where "-" is a don't care. When $h = 1$, either 0x8 or 0x1 are selected. When $h = 0$, 0x4 is selected. The corresponding dichotomy is {{0x8, 0x1},{0x4}}. In this case, function h distinguishes 0x8 from 0x4 and 0x1 from 0x4, covering two seed dichotomies {{0x8},{0x4}} (or {{0x4},{0x8}}) and {{0x1},{0x4}} (or {{0x4},{0x1}}). △

A candidate BS function is generated by assigning each i-set to the ON-set, OFF-set, or DC-set. Hence, the total number of possible BS candidate functions is 3^μ. Nonetheless, some BS candidate functions are interchangeable, i.e., one candidate can be obtained by swapping the ON-set and the OFF-set of another candidate. Our enumeration removes these symmetries. Moreover, in a minimum-length encoding, each candidate must have at least r i-sets in the ON-set and r i-sets in the OFF-set, where r is defined as:

$$r(\mu) = \mu - 2^{\lfloor \log_2(\mu-1) \rfloor}. \tag{6}$$

Candidates that do not satisfy this constraint are eliminated as they cannot encode FS functions. For instance, if μ is a power of 2, then $r = \mu/2$, implying that each candidate must distinguish half of the FS functions against the other half. The number of possible BS candidate functions is given by the following formula depending on μ:

$$\mathcal{E}(\mu) = \frac{1}{2} \cdot \sum_{i=0}^{\mu-2r(\mu)} \left[\binom{\mu}{i} \cdot \sum_{j=r(\mu)}^{\mu-i-r(\mu)} \binom{\mu-i}{j} \right]. \tag{7}$$

Note that when μ is a power of 2 the number of possible BS candidate functions reduces to $\binom{\mu}{\mu/2}/2$.

A limitation of this method is that the number of candidates grows rapidly with increasing column multiplicity. However, we may further reduce the number of BS candidate functions when it is too large. In particular, for an ACD into 6-LUTs the maximum column multiplicity to support is 16 and Equation 7 is maximized for $\mu = 13$ with $91,377$ candidate BS functions. To maintain a reasonable number of candidates to reduce run time, our method does not use the DC-set for problems with $\mu > 8$, lowering the maximum number of candidates to $6,435$. This choice has a marginal effect in quality. This simplification removes the leftmost sum and fixes $i = 0$ in Equation 7.

Each BS candidate function is associated with a cost that depends on the number of variables in its support. The number of variables is computed using a special procedure that

| | 4 | 3 | 3 |
	C9AF	1177	2727
$\{\{0x8\}, \{0x0\}\}$	1	0	1
$\{\{0x8\}, \{0x4\}\}$	1	1	0
$\{\{0x8\}, \{0x1\}\}$	0	1	1
$\{\{0x0\}, \{0x4\}\}$	0	1	1
$\{\{0x0\}, \{0x1\}\}$	1	1	0
$\{\{0x4\}, \{0x1\}\}$	1	0	1

Fig. 4. Covering table to solve the encoding problem.

considers don't cares. Each variable is checked individually. If the incompletely specified BS candidate function remains equivalent when a variable is assigned both constant 0 and constant 1, that variable is not in the functional support and can be removed. Then, a covering table is constructed by having all the pairs of FS functions to be distinguished (seed dichotomies) as rows and the BS candidates as columns. A row-column entry (i, j) is 1 if the BS candidate function of column j distinguishes the seed dichotomy i. A support-minimum solution is computed by solving a minimum-cost covering problem [33]. The solution must cover all the rows while minimizing the cost. We use greedy covering followed by local search to compute a minimum-cost cover. A single iteration of greedy covering extracts one column covering the most non-covered rows while minimizing the cost. The process is iterated until a solution is found. Then, the solution is iteratively improved by replacing one column with another column having a lower cost.

Example 5: Figure 4 shows a covering table reflecting the examples in this section. Each column is a candidate BS function shown as a truth table in hexadecimal format on 4 variables. Each BS candidate function has a cost based on the number of variables on its support (showed above). Each row is a seed dichotomy. An element (i, j) in the table is 1 if the BS_j distinguishes the seed dichotomy i. The best solution with cost 6 takes the second and third columns and leads to two BS functions depending on 3 variables. △

Given a solution, an encoding of the FS functions is obtained by assigning a code $T = t_{M-1} \ldots t_0$, in which each signals t_i corresponds to a selected BS_i candidate.

Example 6: Continuing Example 5, a minimum cover results in $BS_0 = 0x1177$, by putting 0x4 and 0x1 in the ON-set, and $BS_1 = 0x2727$ by putting 0x0 and 0x1 in the ON-set. Both bound sets depend only on 3 variables. Given the BS functions, the encoding of the FS functions assigns the following codes to g_i in Figure 3: $T_{0x8} = 00$, $T_{0x4} = 01$, $T_{0x0} = 10$, and $T_{0x1} = 11$. Finally, the composition function is computed using the FS functions and the selected encoding, resulting in function 0x1048, in hexadecimal format. Consequently, the function has been successfully decomposed using three 4-LUTs. The final result of decomposition is shown in Figure 5, after minimizing the support of BS functions. △

D. Maximizing the shared set

The number of LUTs required to implement the BS functions can be minimized using the shared set. In Section IV-C,
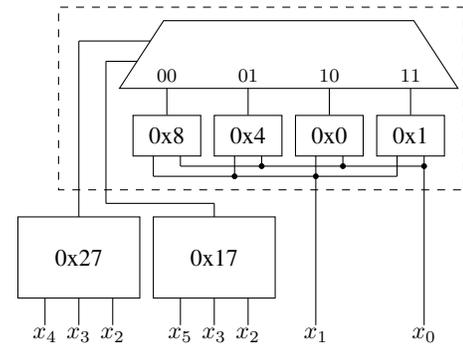

Fig. 5. AC decomposition of the Boolean function 0x880480018414811.

we presented a generic method to find an encoding that minimizes the LUT count and the support size. Alternatively, to check whether a decomposition with $L \in [0, M)$ single-variable functions (or buffers) and $M - L$ non-buffer BS functions exists, our method may enumerate subsets of L out of $N - P$ variables, with a total of $\binom{N-P}{L}$ subsets. For each subset, the method checks if the number of unique FS functions in each cofactor with respect to L variables does not exceed 2^{M-L}. If this is the case, a decomposition with L variables in the shared set exists.

Example 7: Consider the truth table 0xffff0880ffff0000 with $P = 2$ and unique FS functions 0xf, 0x0, and 0x8. Let us check the existence for a shared set when $M = 2$ using $L = 1$. If the most significant variable is in the SS, the truth table is divided into two cofactors 0xffff0880 and 0xffff0000. The number of unique FS functions in the first cofactor exceeds $2^{2-1} = 2$. Hence, the most significant variable cannot be shared. However, the second most significant variable, with cofactors 0xffffffff and 0x08800000, can be shared. △

V. TECHNOLOGY MAPPING WITH ACD

In this section, we leverage the Ashenhurst-Curtis decomposition (ACD) methods described in Section IV to improve the delay of LUT networks. ACD can be used in two ways: 1) as part of LUT mapping, or 2) as a post-mapping resynthesis method to compact logic and decrease the delay. In this work, we focus on the former usage since it has more flexibility and offers good optimization opportunities. While this work does not cover post-mapping resynthesis, its implementation would involve extracting cuts consisting of a few LUTs, computing the cut function as a truth table, and finally performing ACD. If the new implementation is better, it replaces the old one. For an example of how this resynthesis engine could be implemented, we refer the reader to [34]. First, this section discusses how to perform delay-oriented functional decomposition for any number FS variables and BS functions. Then, it describes the integration of ACD in a technology mapper.

A. Delay-oriented ACD

Let us consider a node n in a k-LUT network and a cut C rooted in n that contains leaves in the input sub-network of n. Among all the leaves, some are timing-critical and some are not. Let D be the latest arrival time of a leaf in C. We

use ACD to find an implementation that realizes the function of cut C with delay $D + 1$, when $|C| > k$, assuming a unit-delay model. Specifically, we put the timing-critical leaves of C into the FS and other non-critical ones into the BS or SS. This transformation, when applied on the critical path, may reduce the worst delay of a LUT network.

The ACD-based transformation is performed in two steps. First, our method verifies the existence of a delay-minimizing decomposition. Second, if a decomposition exists, it solves the encoding problem and returns a solution.

1) Checking the existence of a decomposition: Algorithm 1 shows the procedure *evaluate* used to check the existence of an ACD. The algorithm receives the function represented as a truth table tt of a large cut of size N where $N > k$. Set S contains a list of timing-critical variables with delay D. First, the truth table is transformed to have critical variables as the least significant ones since they must be in the FS (at line 3). The proposed approach limits $N - P \leq k$ targeting a two-level decomposition without solving the encoding problem. Hence, the number of variables in the FS must be at least $P \geq N - k$, and $P \geq |S|$ to include all the delay-critical variables (in line 6). For each FS of P_i variables, the column multiplicity value is computed using the method described in Section IV-B, and the smallest one is returned (at line 7). In this case, since delay-critical variables are always part of the FS, $\binom{N}{P_i - |S|}$ different combinations are enumerated. If the configuration with the smallest column multiplicity is implementable using at most $k - P_i$ BS functions, a delay-minimizing ACD exists. In this case, variables in the FS have the delay increase of 1 while other variables have the delay increase of 2 (at line 14). If, on the other hand, a decomposition with P_i does not exist, the function is not decomposable.

The loop in line 6 checks the existence of a decomposition starting with a smaller value of P. Notably, if a decomposition with P does not exist, neither does it exist with $P + 1$, according to Theorem 2. Then, if a decomposition exists, the loop attempts to identify independent shared-set variables (ISS) to add to the free set, according to Theorem 1. Specifically, maximizing the free set to include non-critical variables has multiple benefits. First of all, the decomposition would have a reduced column multiplicity, which simplifies the encoding problem. Additionally, including ISS in the FS may reduce the required time of the associated non-critical signals, facilitating area recovery during technology mapping.

2) Computing the decomposition: After applying *evaluate*, another procedure *decompose* computes the actual decomposition, as described in Section IV-C.

B. LUT mapping with ACD

The methods described in Section V-A have been integrated into an LUT mapping algorithm. State-of-the-art technology mapping typically performs delay minimization followed by multiple iterations to recover area [24]. Each mapping iteration computes k-feasible cuts rooted in nodes of the subject graphs and selects one best cut for each node based on the cost function and slack. Typically, enumerated cuts are k-feasible, meaning they can be implemented using a k-input LUT. In

---

Algorithm 1: ACD evaluation

---
1 Input : Truth table tt, LUT size k, Late vars set S
2 Output: Propagation delay
3 reorder_variables(tt, S);
4 $\mu_{best} \leftarrow \infty$;
5 $\vec{x}_{fs} \leftarrow \emptyset$;
6 for $P_i \leftarrow \max(num\_vars(tt) - k, |S|)$ *to* $k - 1$ **do**
7 $\quad \{\mu, \vec{x}'_{fs}\} \leftarrow$ compute_smallest_multiplicity(tt, P_i, $|S|$);
8 $\quad$ **if** $\mu \leq 2^{k - P_i}$ *and* $\mu < \mu_{best}$ **then**
9 $\quad\quad$ $\mu_{best} \leftarrow \mu$;
10 $\quad\quad$ $\vec{x}_{fs} \leftarrow \vec{x}'_{fs}$;
11 $\quad\quad$ **continue**;
12 $\quad$ **break**;
13 if $\mu_{best} \neq \infty$ **then**
14 $\quad$ **return** compute_propagation_delay(tt, $\vec{x}_{fs}$);
15 return infinite_propagation_delay();

---

our implementation, cut enumeration computes large cuts up to size $k < l \leq 11$, where l is provided by the user. During cut enumeration, the mapper computes cut functions as truth tables. For the non-k-feasible computed cuts, the mapper uses Algorithm 1 to check the existence of a delay-minimizing decomposition into k-LUTs. If a decomposition does not exist, the cut is discarded. If a decomposition exists, the cut delay is computed using the propagation delay returned by Algorithm 1. The area is estimated using column multiplicity. Specifically, to have precise area information, i.e., the number of required LUTs, ACD has to solve the encoding problem and compute the decomposition. However, experimentally, not running the decomposition on the fly reduces the run time considerably with a negligible impact on the final area. The area is estimated conservatively, neglecting the existence of a shared set, i.e., $Area = \lceil \log_2 \mu \rceil + 1$.

The mapper uses l-feasible cuts with ACD in the delay mapping pass, while it uses k-feasible cuts in the following area recovery. Note that area recovery aims at improving the solution over non-critical paths and can re-use the best cuts from the previous passes, while assuring that the required times are met. After the last mapping pass, a cover is generated consisting of k- and l-feasible cuts. At this stage, the mapper decomposes non-k-feasible cuts into k-LUTs.

VI. MAPPING INTO IN-SLICE CASCADES

As mentioned in the introduction, the delay in the modern FPGAs is often dominated by that of programmable interconnect. To reduce the need for signal routing, one approach modifies the FPGA architecture to include non-routable connections between LUTs. For instance, recent FPGAs produced by AMD have configurable logic blocks (CLBs) divided into *slices*. A slice contains 8 LUTs that can be used independently, with external routing or using internal *cascade* connections [10]. Specifically, a slice LUT LUT_i, with $0 \leq i < 8$, may connect one of its 6 inputs to LUT_{i-1}, forming a cascade structure. An in-slice cascade connection is 10 to 40 times faster than standard interconnect, which helps delay optimization.

Although in-slice non-routable connections are available, LUT networks generated by the traditional LUT mapping do

not use them efficiently. This is because a placement algorithm may fully leverage non-routable connections only for LUTs on the critical path with one critical fanin. In practice, however, LUTs on the critical path tend to have multiple critical fanins, making it hard for the placement algorithm to utilize cascade structures.

An efficient way to leverage cascade connections is to generate mappings of LUTs into cascades during technology mapping. LUT cascades can be generated by decomposing large non-k-feasible functions. In the section, we use the ACD method of Section IV to compute decompositions into specific structures of two LUTs, called "kk" decomposition. Contrary to previous approaches [11], our approach is not based on a heuristic and may support more than one variable in the shared set. Specifically, it always finds a solution if it exists.

A. ACD into two LUTs

A decomposition into two LUTs is a special type of ACD with a single BS function and possibly multiple SS variables. Since BS functions are limited to one, the problem has a lower complexity than the generic case. Here we propose a dedicated algorithm to solve this problem more efficiently.

For a truth table on N variables, a "kk" decomposition may exist for $N < 2 \cdot k$. According to Theorems 1 and 2, it is sufficient to test the decomposition for $P = N - k$, when allowing for multiple variables in the shared set. Specifically, this is the minimum number of variables to have a k-feasible bound set and a decomposition. Note that a decomposition with $P < N - k$ (or $N - P > k$) may exist only if there are at least y independent variables in the shared set, such that $P + y = N - k$. Since, by Theorem 1, ISS variables can always be moved into the free set, and, by Theorem 2 a smaller free set has more solutions than a larger one, $P = N - k$ is the only necessary FS size to check.

Algorithm 2 shows a sequence of steps to perform a decomposition into two LUTs. The algorithm takes as inputs a truth table tt, the number of its support variables N, and the LUT size k. First, P and the permutation vector $Perm$ are initialized. Vector $Perm$ is necessary to track the order of the variables during the enumeration of combinations, compared to the original one, and to compute the next combination. A loop iterates on all the possible P combinations of N. The method next_combination (at line 12) computes a new combination from the previous one by swapping one variable in the FS with one in the BS. The returned truth table reflects the new variable order. The column multiplicity μ is computed for the truth table tt (at line 6). If $\mu = 2$, a decomposition exists with one BS function. Since the structure is limited to one BS function, for $\mu > 2$ the method searches for SS variables. First, L_{min} is computed to minimize the number of shared variables. Then, the algorithm searches for a shared set of L elements, employing the techniques of Section IV-D. The search for a shared set is performed for $L_{min} \leq L < k - P$, which also allows for non-minimum-length encodings. If a shared set exists, the corresponding decomposition is returned. Otherwise, if the conditions in the for loop are not met, the function is not decomposable into 2 LUTs.

Algorithm 2: ACD into two LUTs

---

1 Input : Truth table tt, number of variables N, LUT size k
2 Output: Decomposition if it exists
3 $P \leftarrow N - k$;
4 $Perm \leftarrow \{0, 1, 2, \ldots, N - 1\}$;
5 for $\binom{N}{P}$ *iterations* **do**
6 $\mu \leftarrow$ compute_multiplicity(tt, P);
7 $L_{min} \leftarrow \lceil \log_2 \mu \rceil - 1$;   ▷ Required variables in SS
8   **if** $P + L_{min} < k$ **then**
9 $\vec{x}_{ss} \leftarrow$ compute_shared_set(tt, N, P, k, L_{min});
10   **if** $P + |\vec{x}_{ss}| < k$ **then**
11   **return** decompose(tt, N, P, k, $Perm$, $\vec{x}_{ss}$);
12 $tt \leftarrow$ next_combination(tt, N, P, $Perm$);
13 return not decomposable;

In case of an implementation constraining the maximum number of variables in the SS, Algorithm 2 is modified to additionally explore different sizes P, similarly to Algorithm 1. This is because Theorem 2 is not valid when limiting the maximum number of BS functions and SS variables because it constraints the maximum value of encoding M. Hence, when $\lceil \log_2(\mu) \rceil > M_{max}$ there might be ISS variables to include in the FS to make $\lceil \log_2(\mu') \rceil \leq M_{max}$.

B. Mapping into LUT structures

We follow the method proposed in [11] for mapping into LUT structures. Specifically, the LUT mapper performs cut enumeration using cuts up to size l with $k < l \leq 2 \times k$, derives their functions as truth tables, and checks if the functions are decomposable into a "kk" structure. If a function is decomposable, the area and delay are assigned based on a given LUT library. If the function is not decomposable, the cut is ignored. An LUT library specifies the area and delay of an LUT based on its size. Similarly to Section V-B, the mapper begins by minimizing delay, followed by several iterations of area recovery. Contrarily to Section V-B, the mapper uses ACD decomposition of l-feasible cuts during all mapping iterations.

VII. EXPERIMENTS

This section presents an experimental evaluation of the proposed LUT mapping with ACD. First, we evaluate the ACD-based algorithms proposed in this paper on practical functions extracted from open-sources hardware designs. Then, we evaluate ACD in the context of delay-driven LUT mapping. Finally, we present the results of mapping into LUT cascade structures. While the experiments are reported for 6-input LUTs, similar improvements have been obtained also for 4-input LUTs.

The proposed methods have been implemented and are available in the open-source logic synthesis framework *ABC* [35]. For our experiments, we use the EPFL combinational benchmark suite [36] containing several circuits provided as *and-inverter graphs* (AIGs). The baseline has been obtained using the following script "`dfraig; resyn; resyn2; resyn2rs; if -y -K 6; resyn2rs;`" in ABC, which perform a high-effort size and depth AIG optimization. In particular, it combines SAT sweeping [37],

TABLE I
DECOMPOSITION SUCCESS RATIO INTO TWO 6-LUTs FOR PRACTICAL FUNCTIONS USING DIFFERENT ACD METHODS.

| ACD type | 7 vars (41071) | | 8 vars (107466) | | 9 vars (195602) | | 10 vars (313649) | | 11 vars (404991) | |
	Success (%)	Time(s)	Success (%)	Time(s)	Success (%)	Time(s)	Success (%)	Time(s)	Success (%)	Time(s)
S66 [11]	84.18%	0.60	69.24%	2.57	52.13%	4.99	37.36%	6.99	19.14%	9.79
lutpack [19]	98.34%	20.39	83.47%	64.37	69.92%	154.38	48.95%	334.79	26.87%	897.55
J66 1-SS	97.30%	0.28	82.23%	1.41	74.24%	4.20	63.06%	9.39	32.88%	16.43
J66 M-SS	99.82%	0.30	92.94%	3.08	84.71%	9.92	63.06%	9.73	32.88%	16.58

scripts for delay-oriented AIG optimization [21], and lazy man's logic synthesis [38], which is the most aggressive depth minimization for AIGs in ABC. The experiments have been conducted on an Intel i5 quad-core 2GHz on MacOS. The results have been verified using combinational equivalent checkering in ABC.

We extended the LUT mapper *if* (and $\&if$) in ABC to perform ACD, as discussed in Sections V and VI. The following commands are used in the experiments:

- dch (-f): computes structural choices used to mitigate the structural bias [4], where -f stands for "fast";
- if -K 6: performs delay-oriented technology mapping with choices into 6-LUTs using 6-feasible cuts;
- if -s -S 66 -K 8: performs delay-oriented technology mapping using 8-feasible cuts and decomposes logic for minimal delay into two 6-LUTs using a SAT-based formulation;
- if -Z 6 -K 8: performs technology mapping into 6-LUTs using the proposed delay-oriented implementation of ACD described in Section V on 8-feasible cuts;
- if -S 66: performs technology mapping based on a given LUT library and packs logic into a structure composed of two 6-LUTs using the ACD method from [11];
- if -J 66: performs technology mapping based on a given LUT library and packs logic into a structure composed of two 6-LUTs using the ACD method described in Section VI;
- st: derives an AIG from an LUT network.

A. Decomposition success rate

In this experiment, we evaluate the performance of ACD in decomposing functions by comparing it against other implementations of Boolean decomposition in ABC. Specifically, we test the number of functions that can be successfully decomposed and the run time needed. We run this experiment on *practical functions*, i.e., functions collected in hardware designs and benchmarks, which include fully-decomposable, partially-decomposable, and non-decomposable functions. Practical functions tend to be much less than all possible functions since designs are never completely random. We extract practical functions from the EPFL benchmarks [36] by recording all the functions encountered during cut enumeration in a technology mapper. Since the number of practical functions can be large, we classify them into $\mathcal{NPN}$-equivalence classes employing the heuristic sifting algorithm [39].

Table I shows the percentage of decomposable functions and the run time for different methods and support sizes.

For instance, the first column contains results for decomposing practical 7-input functions, where (41071) indicates the number of unique functions collected after computing $\mathcal{NPN}$ canonical forms. Each row of the table shows one ACD method. The first row S66 presents the state-of-the-art method in [11] to decompose into a LUT structure composed of two 6-LUTs. Note that S66 supports no more than one variable in the shared set. The next approach *lutpack* [19][1] performs a heuristic ACD using DSD and the Shannon expansion, supporting up to 3 variables in the shared set. Finally, we present two variants of the decomposition method into LUT structures composed of two 6-LUTs described in Section VI, denoted J66. J66 1-SS uses up to one variable in the SS to better compare against S66. Meanwhile, J66 M-SS has no restrictions on the number of SS variables.

Table I shows that the approaches described in this paper outperform state-of-the-art. In particular, J66 1-SS has a significantly better success rate in all columns and better run time up to 9-input functions, compared to S66. Notably, while searching for a decomposition with the same characteristics, J66 1-SS always finds a solution if it exists (under the 1-SS limitation), while S66 does not always find it because it uses heuristics. This leads to an improvement in success rate that peaks at 25.7%. This table shows the potential of the methods proposed in this work, which can outperform state-of-the-art in quality and run time. J66 M-SS further improves the results for functions between 7 and 9 inputs, with an improvement that peaks at 32.58%, compared to S66.

Regarding the run time, while Table I shows that S66 is generally faster than J66, J66 is, on average, faster for decomposable functions and considerably slower for non-decomposable ones. In fact, J66 enumerates all the possible free sets to find a solution if it exists, while S66 limits the exploration to a smaller subspace.

B. Decomposition success rate for delay optimization

We extend the previous experiment to evaluate delay minimization using the proposed ACD methods. This experiment tests the success rate of a delay-minimal decomposition for practical functions given delay-critical variables required to be in the free set. Informally, for delay-critical variables with delay D, this experiment checks the existence of a decomposition with delay $D + 1$. The other variables are considered to have delay $D - 1$. We only consider J66 M-SS and generic ACD since other known methods do not perform delay minimization

---

[1]We modified lutpack in ABC to perform only the decomposition required by the experiment without the overhead of the resynthesis engine.

TABLE II
DECOMPOSITION SUCCESS RATIO INTO 2 LEVELS OF 6-LUTS FOR
PRACTICAL FUNCTIONS GIVEN LATE ARRIVING VARIABLES.

N late	ACD type	7 vars	8 vars	9 vars	10 vars	11 vars
0	lutpack [19]	99.01%	88.25%	84.65%	75.25%	26.87%
	J66 M-SS	99.82%	92.94%	84.71%	63.06%	32.88%
	Generic	100.00%	100.00%	98.05%	90.20%	32.88%
1	J66 M-SS	96.59%	79.60%	61.51%	37.35%	16.54%
	Generic	100.00%	100.00%	97.57%	83.23%	16.54%
2	J66 M-SS	86.22%	59.78%	39.28%	23.74%	10.95%
	Generic	100.00%	100.00%	94.19%	66.56%	10.95%
3	J66 M-SS	65.11%	36.37%	21.25%	13.78%	6.96%
	Generic	93.78%	86.03%	76.82%	44.51%	6.96%
4	J66 M-SS	36.96%	17.00%	8.62%	7.21%	4.43%
	Generic	54.55%	40.42%	25.45%	23.70%	4.43%
5	J66 M-SS	14.52%	5.42%	2.96%	2.84%	2.61%
	Generic	14.52%	5.42%	2.96%	2.84%	2.61%

using input arrival times. We show *lutpack* [19] only for the first row to perform a 2-level decomposition, without limiting the number of LUTs. For each function, we randomly generate up to 10 unique sets of delay-critical variables and test the decomposition for each one of them.

Table II shows the success rate of decomposing practical functions based on the number of delay-critical variables, shown in column "N late". Generic ACD has a high success rate in most cases. Limitations occur when the number of delay-critical variables exceeds 3 or the number of variables in the support is 10 or more. Generally, the decomposition of 11-input functions is rare. However, many 10 input functions are still decomposable. Furthermore, the table highlights the advantages of using multiple BS functions, with a success rate difference between J66 and generic that peaks at 55.57% for 9-input functions, given 3 delay-critical variables. Thus, in this case, it is 55% more likely to find a solution to a delay-driven decomposition problem if we consider the most general two-level ACD formulation, compared to the case when only J66 is used.

C. Delay-driven LUT mapping

Table III compares four technology mapping strategies for delay minimization during mapping into 6-LUTs, assuming a unit-delay model. Each strategy takes the baseline as an input and computes structural choices before mapping. Structural choices have not been used for the benchmark *hyp* due to a known bug in ABC. The proposed method is compared against standard LUT mapping and mapping into LUT structures. In the rightmost column, command *ACD* denotes the sequence "dch; if -Z 6 -K 8". We do not compare against [11] and [19] because those methods perform only area-oriented ACD. Furthermore, we do not compare against the recent mapper with gate decomposition based on bin-backing [40]. Nevertheless, the mapper in [40] can improve the delay of if by only 0.31% on average.

Mapping into LUT structures "66" composed of two 6-LUTs, which is based on a limited version of structural ACD, reduces depth by 1.04% and the area by 2.57% on average, at

the cost of increasing the number of edges by 2.57%. The proposed LUT mapping with ACD improves the depth of the LUT network by 7.52% on average while increasing the number of LUTs and edges by 8.13% and 7.87%, respectively.

Note that most of the improvements are due to the first 10 benchmarks since others are already close to their optimal depth. For 4 of them, the delay reduction exceeds 20% and is up to 27.27%. Practically, part of the area increase can be reduced by area recovery [19], [41], [42], using delay relaxation, or by an additional mapping step applied after ACD. The rightmost strategy performs the latter option. The LUT count and edge count are reduced considerably, leading to an area improvement of 2.20%, compared to traditional technology mapping with choices. Also, the logical depth further decreases up to 54.55%. To achieve this, the LUT network after ACD is used as a structural choice to improve the next round of mapping because choices extracted from mapping with ACD are more structurally suited to delay-oriented mapping, compared to the original AIG. Moreover, structural choices help reduce the area on the non-critical paths. Note that a second mapping round does not give practical benefits if applied after the default LUT mapper (leftmost column) since the network after deriving the AIG is structurally similar to the baseline. Furthermore, benchmark *hyp* is noticeably improved by remapping both in area and delay, although it does not use structural choices. Regarding the run time, mapping with ACD is much faster than mapping into LUT structures while being more general.

D. EPFL synthesis competition

This experiment shows that ACD-based LUT mapping can improve well optimized LUT networks, resulting in best known results for 4 benchmarks in the ongoing EPFL synthesis competition. The previous best results were obtained using a portfolio of heavy logic optimization applied to various representations, such as AIGs and LUT networks. In recent years, results have been further improved using *design-space exploration* (DSE) techniques that incrementally generate optimization scripts and visit multiple points of the design space. Examples of these methods are: Bayesian optimization [44], reinforcement learning [45], machine learning, and other heuristic approaches.

We compete in the best delay competition by using standard delay-oriented scripts in ABC and LUT mapping with ACD. We do not use DSE to show that the proposed method outperforms or gets close to the best results in the competition. We obtain the optimized AIGs by repeatedly running the script used in the baseline of Table III along with additional delay-oriented AIG commands in ABC. For the resulting AIGs, we compare traditional LUT mapping with choices and LUT mapping with ACD. Notably, results by the traditional mapper are quite far from the best results. This observation shows that our technology-independent optimization finds worse AIGs than those used to obtain the best results, as expected. However, LUT mapping with ACD matches or improves the depth for almost all the benchmarks. The improved benchmarks are *hyp*, *log2*, *multiplier*, and *square*. Remarkably, our method reduces

TABLE III
COMPARISON OF DELAY-DRIVEN LUT MAPPING, LUT MAPPING TO "66" STRUCTURE, AND LUT MAPPING USING ACD.

Benchmark	dch; if -K 6				dch; if -s -S 66 -K 8				dch; if -Z 6 -K 8				ACD; st; dch -f; if -K 6			
	LUTs	Edges	Depth	Time (s)	LUTs	Edges	Depth	Time (s)	LUTs	Edges	Depth	Time (s)	LUTs	Edges	Depth	Time (s)
adder	363	1433	22	0.18	362	1465	20	0.28	383	1519	16	0.20	353	1518	10	0.39
bar	1664	9344	4	0.44	1664	9344	4	0.57	1664	9344	4	0.47	1006	5274	4	0.76
div	8618	32394	406	6.62	9107	33665	397	13.42	11644	44496	326	7.16	9068	39167	271	21.19
hyp	58393	239097	1864	5.43	61701	247699	1840	31.82	65615	264998	1396	11.13	61769	263254	1034	19.76
log2	9712	43562	58	17.05	10172	44943	58	30.06	10313	46365	56	17.81	9429	42533	57	39.09
max	831	3804	14	0.37	840	3668	14	0.63	1211	5578	12	0.42	871	4277	11	1.39
multiplier	7383	34137	36	6.01	7334	32781	36	12.11	7693	35798	33	6.82	6800	31705	31	13.32
sin	1928	8445	30	1.31	1948	8463	30	4.94	2052	8913	29	1.50	1830	8178	30	2.91
sqrt	7515	29573	663	4.17	7972	30610	638	12.66	10156	38558	519	4.73	9292	36030	476	8.77
square	4122	17319	23	1.98	4165	17547	22	3.91	4107	17924	18	2.22	4118	18285	14	5.15
arbiter	1833	8982	6	1.64	1879	8836	6	2.02	1850	8987	6	1.70	2037	8780	6	3.33
cavlc	137	707	4	0.13	104	491	4	0.56	137	707	4	0.15	123	655	4	0.20
ctrl	30	133	2	0.07	28	127	2	0.08	30	133	2	0.08	29	126	2	0.08
dec	287	684	2	0.09	287	1404	2	0.1	287	684	2	0.10	284	816	2	0.12
i2c	312	1360	3	0.16	306	1316	3	0.36	319	1378	3	0.19	297	1329	3	0.27
int2float	52	258	3	0.08	46	205	3	0.18	52	258	3	0.09	50	251	3	0.11
mem_ctrl	11037	48812	18	10.24	10830	46368	18	31.67	11232	49483	17	11.40	10398	45793	16	20.57
priority	178	725	6	0.11	182	736	6	0.18	185	736	6	0.12	171	698	6	0.17
router	89	285	4	0.09	61	283	4	0.14	92	290	4	0.09	89	279	4	0.12
voter	1838	8596	13	2.23	1784	8624	13	4.14	1838	8583	13	2.32	1777	8426	13	4.82
Reduction					2.57%	-2.57%	1.04%		-8.13%	-7.87%	7.52%		2.20%	-0.30%	12.39%	
Total				58.40				149.83				68.70				142.52

TABLE IV
LUT MAPPING IN THE EPFL SYNTHESIS COMPETITION.

Benchmark	Best [43]		dch -f; if -K 6		dch -f; if -Z 6 -K 10	
	LUTs	Depth	LUTs	Depth	LUTs	Depth
adder	347	5	360	6	445	5
bar	512	4	512	4	512	4
div	25318	175	23461	192	31526	175
hyp	182723	483	122394	511	154903	473
log2	8617	52	8778	60	9613	51
max	1114	6	1113	7	1250	6
multiplier	7785	25	6839	28	6903	25
sin	680530	10	1820	33	2379	27
sqrt	29593	162	30945	172	41626	156
square	3732	10	4189	11	4275	10

the depth of *hyp* by 10 levels, compared to state-of-the-art while also reducing area by 15%. In the benchmark *multiplier*, our result matches the depth but improves the number of LUTs. Benchmark *sin* is the only one where there is a large gap compared to the best result. It is likely that the best result for *sin* requires significant logic duplication not performed in our synthesis flow.

Unlike many other methods used to produce the best results, our results in Table III are obtained directly by LUT mapping without post-mapping optimization. For instance, if we use LUT resubstitution, the area of *multiplier* is further reduced to 6499 nodes. Even better results are expected by integrating ACD-based LUT mapping into a DSE flow.

E. Mapping into LUT structures

In this experiment, we perform technology mapping into LUT structures by leveraging non-routable cascade connections of LUTs in FPGA architectures. Specifically, motivated by the high cost of routing, we assume that a 6-LUT and a cascade of two 6-LUTs both have unit delay. A more precise

model would assign propagation delay of about 1.2 to the signals in the bound set of a LUT cascade and unit delay to the signals connected to the composition function. However, the mapper in [11] only supports a fixed delay assignment to all the signals. Hence, we assume the delay of a cascade to be unitary to not penalize the quality of mapping into LUT structures. We run all the mappers with the same parameters to perform minimal-delay mapping. Mappers running ACD use cuts up to 10 inputs.

Table V compares traditional LUT mapping with choices, the LUT structure mapping [11], and the proposed method described in Section VI supporting 1 (1-SS) or multiple (M-SS) shared set variables. S66 improves the traditional mapper by 30.74% in delay while increasing area and the number of edges. For many benchmarks, the area increases due to logic duplication to minimize delay. Notably, J66 1-SS considerably improves all the metrics, compared to S66. The improvement comes from the better success rate of the decomposition shown in Table I. Moreover, J66 M-SS achieves further improvement, compared to S66, reducing the average delay, area, and edge count by 6.22%, 3.82%, and 3.09%, respectively, with a faster run time. Remarkably, for designs with a similar delay to the traditional mapper, J66 achieves a large reduction in the number of LUTs and edges. This is because J66 successfully mitigates structural bias. For instance, for benchmark *int2float*, J66 M-SS reduces the number of LUTs by 27%. For the same benchmark, S66 reduces the number of LUTs only by 1.92%. Similar improvements are also observed for all the benchmarks when performing area-oriented mapping, instead of delay-oriented mapping. Another interesting benchmark is *cavlc*, where multiple shared set variables significantly improve the delay, area, and edge count.

Finally, while S66 is generally faster than J66 for large functions, the mapping time of J66 is better than S66. This

TABLE V
COMPARISON OF DELAY-DRIVEN LUT MAPPING AND MULTIPLE ACD-BASED MAPPING INTO "66" CASCADE STRUCTURES.

Benchmark	dch; if -K 6				dch; if -S 66 [11]				dch; if -J 66 1-SS				dch; if -J 66 M-SS			
	LUTs	Edges	Delay	Time (s)	LUTs	Edges	Delay	Time(s)	LUTs	Edges	Delay	Time(s)	LUTs	Edges	Delay	Time(s)
adder	363	1433	22	0.18	352	1521	13	0.85	356	1552	13	0.52	354	1550	13	0.83
bar	1664	9344	4	0.44	1664	8320	3	1.34	1664	8320	3	0.75	1664	8320	3	0.78
div	8618	32394	406	6.62	11555	46558	266	34.87	11071	45711	251	27.54	11298	47587	248	30.87
hyp	58393	239097	1864	5.43	65987	274992	1144	270.03	65352	274000	1082	161.75	65175	273434	1076	183.77
log2	9712	43562	58	17.05	12813	59950	42	73.19	12526	58798	40	54.03	12409	59528	39	71.87
max	831	3804	14	0.37	1177	6162	9	1.77	1113	5448	9	1.31	1113	5448	9	1.44
multiplier	7383	34137	36	6.01	8898	42566	25	46.10	8861	42005	25	31.27	8645	43556	24	36.15
sin	1928	8445	30	1.31	2620	12074	22	12.45	2461	11125	21	9.39	2400	10977	21	13.17
sqrt	7515	29573	663	4.17	9510	37809	423	42.21	9109	37396	403	24.86	9441	38373	398	32.31
square	4122	17319	23	1.98	4299	19677	15	11.75	4290	19843	14	8.07	4299	19972	14	12.65
arbiter	1833	8982	6	1.64	2000	9481	4	2.71	1992	9834	4	2.53	1992	9834	4	2.50
cavlc	137	707	4	0.13	125	645	3	0.49	124	639	3	0.43	110	565	2	0.54
ctrl	30	133	2	0.07	28	131	2	0.08	28	133	1	0.09	28	133	1	0.09
dec	287	684	2	0.09	512	2304	1	0.11	512	2304	1	0.12	512	2304	1	0.12
i2c	312	1360	3	0.16	327	1530	2	0.49	319	1478	2	0.41	306	1433	2	0.45
int2float	52	258	3	0.08	51	257	2	0.17	42	216	2	0.17	38	191	2	0.20
mem_ctrl	11037	48812	18	10.24	11666	52725	13	59.20	11247	51109	13	48.04	11019	50726	13	56.43
priority	178	725	6	0.11	175	761	4	0.21	176	768	4	0.28	176	768	4	0.28
router	89	285	4	0.09	65	305	3	0.15	65	306	3	0.19	65	303	3	0.21
voter	1838	8596	13	2.23	2133	10793	10	7.22	2068	10082	10	5.58	2053	10081	10	7.87
Reduction					-13.65%	-27.62%	30.74%		-10.73%	-24.52%	34.29%		-9.44%	-24.00%	35.86%	
Total				58.40				565.39				377.33				452.53

is because J66 is faster when applied to frequently appearing decomposable functions and slower when applied to non-decomposable functions. After all, it uses more effort to find a solution. For instance, on the benchmark *sqrt*, which has a considerable run time difference between S66 and J66, only 2.93% of all cuts are not decomposable by J66, against an 11.45% of S66. Moreover, only 10.35% of 10-input cuts are not decomposable by J66 M-SS, while 39.48% are not decomposable by S66. Finally, run time could be further reduced by taking advantage of GPU-based LUT mapping implementations [46].

VIII. CONCLUSION

This work proposes a novel formulation of Ashenhurst-Curtis decomposition (ACD) to enable efficient technology mapping and post-mapping resynthesis. The algorithm is truth-table-based and flexible in terms of the sizes of the free set, bound set, and shared set, which makes it well-suited for delay optimization. We have shown that our Boolean decomposition improves state-of-the-art in decomposition quality with a competitive run time. We have implemented and integrated the proposed method into a delay-driven LUT mapper. The experiments show that LUT mapping with ACD can improve the average delay by 12.39%, compared to traditional structural LUT mapping with choices. Furthermore, the proposed approach has found 4 new best results in the EPFL synthesis competition. Finally, we applied ACD to perform mapping into LUT cascade structures, outperforming state-of-the-art in all metrics.

The findings of this work have impact beyond technology mapping. LUT mappers are key in design-space exploration engines and in various optimization flows, for example, in those used for standard cells [47]. Hence, the methods pro-posed in this paper may significantly improve the quality of logic synthesis tools, especially for delay optimization.

REFERENCES

[1] J. Cong and Y. Ding, "FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *Trans. CAD*, vol. 13, no. 1, pp. 1–12, 1994.

[2] A. H. Farrahi and M. Sarrafzadeh, "Complexity of the lookup-table minimization problem for FPGA technology mapping," *IEEE Trans. CAD*, vol. 13, no. 11, pp. 1319–1332, 1994.

[3] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, "Logic decomposition during technology mapping," *Trans. CAD*, vol. 16, no. 8, pp. 813–834, 1997.

[4] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping," in *Proc. ICCAD*, 2005, pp. 519–526.

[5] R. Francis, J. Rose, and K. Chung, "Chortle: a technology mapping program for lookup table-based field programmable gate arrays," in *DAC*, 1990, pp. 613–619.

[6] G. Chen and J. Cong, "Simultaneous logic decomposition with technology mapping in FPGA designs," in *Proc. FPGA*, 2001, p. 48–55.

[7] C. Legl, B. Wurth, and K. Eckl, "A boolean approach to performance-directed technology mapping for LUT-based FPGA designs," in *DAC*, 1996, pp. 730–733.

[8] J. Cong and Y.-Y. Hwang, "Structural gate decomposition for depth-optimal technology mapping in LUT-based FPGA design," in *DAC*, 1996, pp. 726–729.

[9] V. Manohararajah, S. D. Brown, and Z. G. Vranesic, "Heuristics for area minimization in LUT-based FPGA technology mapping," *IEEE Trans. CAD*, vol. 25, no. 11, pp. 2331–2340, 2006.

[10] "AMD versal CLB documentation." [Online]. Available: https://docs.amd.com/r/en-US/am005-versal-clb/Look-Up-Table

[11] S. Ray, A. Mishchenko, N. Een, R. Brayton, S. Jang, and C. Chen, "Mapping into LUT structures," in *Proc. DATE*, 2012, pp. 1579–1584.

[12] R. L. Ashenhurst, "The decomposition of switching functions," in *Proc. Int. Symp. Theory Switch.*, 1957, pp. 74–116.

[13] J. P. Curtis, *A New Approach to the Design of Switching Circuits*. D. Van Nostrand, 1962.

[14] J. P. Roth and R. M. Karp, "Minimization over boolean graphs," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 227–238, 1962.

[15] C. Legl, B. Wurth, and K. Eckl, "Computing support-minimal subfunctions during functional decomposition," *Trans. VLSI*, vol. 6, no. 3, pp. 354–363, 1998.

[16] R. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.

[17] M. Perkowski, M. Marek-Sadowska, L. Jozwiak, T. Luba, S. Grygiel, M. Nowicka, R. Malvi, Z. Wang, and J. Zhang, "Decomposition of multiple-valued relations," in *Proc. Inter. Symp. on Mult.- Valued Logic*, 1997, pp. 13–18.

[18] N. Vemuri, P. Kalla, and R. Tessier, "BDD-based logic synthesis for LUT-based FPGAs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 7, no. 4, p. 501–525, 2002.

[19] A. Mishchenko, R. Brayton, and S. Chatterjee, "Boolean factoring and decomposition of logic networks," in *Proc. ICCAD*, 2008, pp. 38–44.

[20] A. Mishchenko, R. Brayton, and S. Jang, "Global delay optimization using structural choices," in *Proc. FPGA*, 2010, p. 181–184.

[21] A. Mishchenko and R. Brayton, "Scalable logic synthesis using a simple circuit structure," in *Proc. IWLS*, 2006.

[22] Bertacco and Damiani, "The disjunctive decomposition of logic functions," in *Proc. ICCAD*, 1997, pp. 78–82.

[23] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," in *Proc. FPGA*, 1999.

[24] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts," in *Proc. ICCAD*, 2007.

[25] V. N. Kravets and K. A. Sakallah, "Constructive multi-level synthesis by way of functional properties," Ph.D. dissertation, University of Michigan, 2001.

[26] T. Stanion and C. Sechen, "Quasi-algebraic decompositions of switching functions," in *in Advanced Res. VLSI*, 1995, pp. 358–367.

[27] C. Yang and M. Ciesielski, "BDS: a BDD-based logic optimization system," *IEEE Trans. CAD*, vol. 21, no. 7, pp. 866–876, 2002.

[28] V. Bertacco and M. Damiani, "Boolean function representation based on disjoint-support decompositions," in *Proc. Int. Conf. on Comp. Design*, 1996, pp. 27–32.

[29] Y.-T. Lai, M. Pedram, and S. Vrudhula, "BDD based decomposition of logic functions with application to FPGA synthesis," in *DAC*, 1993.

[30] A. Mishchenko and T. Sasao, "Encoding of boolean functions and its application to LUT cascade synthesis," in *Proc. IWLS*, 2002.

[31] G. De Micheli, R. Brayton, and A. Sangiovanni-Vincentelli, "Optimal state assignment for finite state machines," *Trans. CAD*, vol. 4, no. 3, pp. 269–285, 1985.

[32] T. Villa and A. Sangiovanni-Vincentelli, "NOVA: state assignment of finite state machines for optimal two-level logic implementation," *Trans. CAD*, vol. 9, no. 9, pp. 905–924, 1990.

[33] S. Yang and M. Ciesielski, "Optimum and suboptimum algorithms for input encoding and its relationship to logic minimization," *Trans. CAD*, vol. 10, no. 1, pp. 4–12, 1991.

[34] A. Mishchenko, S. Chatterjee, and R. Brayton, "Fast boolean matching for LUT structures," EECS Dep., UC Berkeley, Tech. Rep., 2007.

[35] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Eds., 2010. [Online]. Available: https://github.com/berkeley-abc/abc

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

[37] A. Mishchenko, S. Chatterjee, and R. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification," EECS Dep., UC Berkeley, Tech. Rep., 2005.

[38] W. Yang, L. Wang, and A. Mishchenko, "Lazy man's logic synthesis," in *Proc. ICCAD*, 2012, p. 597–604.

[39] M. Soeken, A. Mishchenko, A. Petkovska, B. Sterin, P. Ienne, R. K. Brayton, and G. De Micheli, "Heuristic NPN classification for large functions using AIGs and LEXSAT," in *Theory and Applications of Satisfiability Testing*, N. Creignou and D. Le Berre, Eds., 2016.

[40] L. Fan and C. Wu, "FPGA technology mapping with adaptive gate decomposition," in *Proc. FPGA*, 2023, p. 135–140.

[41] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang, "Scalable don't-care-based logic optimization and resynthesis," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 4, 2011.

[42] B. Schmitt, A. Mishchenko, and R. Brayton, "SAT-based area recovery in structural technology mapping," in *Proc. ASP-DAC*, 2018, pp. 586–591.

[43] "EPFL synthesis competition best results [2023]." [Online]. Available: https://github.com/lsils/benchmarks/tree/v2023.1/best_results

[44] A. Grosnit, C. Malherbe, R. Tutunov, X. Wan, J. Wang, and H. B. Ammar, "BOiLS: Bayesian optimisation for logic synthesis," in *DATE*, 2022, pp. 1193–1196.

[45] J. Yuan, P. Wang, J. Ye, M. Yuan, J. Hao, and J. Yan, "EasySO: exploration-enhanced reinforcement learning for logic synthesis sequence optimization and a comprehensive RL environment," in *ICCAD*, 2023, pp. 1–9.

[46] T. Liu, L. Chen, X. Li, M. Yuan, and E. F. Y. Young, "Finemap: A fine-grained GPU-parallel LUT mapping engine," in *Proc. ASP-DAC*, 2024, p. 392–397.

[47] W. L. Neto, L. Amarú, V. Possani, P. Vuillod, J. Luo, A. Mishchenko, and P.-E. Gaillardon, "Improving LUT-based optimization for ASICs," in *Proc. DAC*, 2022.

Alessandro Tempia Calvino received a B.S. degree in Computer Engineering from the Politecnico di Torino, Turin, Italy, in 2017, and an M.S. degree in Computer Engineering from the Politecnico di Torino, in 2020, and Télécom Paris, Paris, France, in 2021. He is currently pursuing a Ph.D. degree in Computer Science at EPFL, Lausanne, Switzerland in the Integrated Systems Laboratory. His research interests include design automation, logic synthesis, and emerging technologies.

Giovanni De Micheli is Professor and Director of the Integrated Systems Laboratory at EPFL Lausanne, Switzerland. He is a Fellow of ACM, AAAS and IEEE, a member of the Academia Europaea and an International Honorary member of the American Academy of Arts and Sciences. His current research interests include several aspects of design technologies for integrated circuits and systems, such as synthesis for emerging technologies. He is member of the Scientific Advisory Board of IMEC and STMicroelectronics. Prof. De Micheli is the recipient of the 2022 ESDA-IEEE/CEDA Phil Kaufman Award, the 2019 ACM/SIGDA Pioneering Achievement Award, and several other awards.

Alan Mishchenko received the M.S. degree from the Moscow Institute of Physics and Technology, Moscow, Russia, in 1993 and the Ph.D. degree from the Glushkov Institute of Cybernetics, Kiev, Ukraine, in 1997. In 2002, he joined the EECS Department, University of California at Berkeley, Berkeley, CA, USA, where he is currently a Full Researcher. His current research interests include computationally efficient logic synthesis, formal verification, and machine learning.

Robert Brayton received his Ph.D. degree in mathematics from MIT in 1961. He was a member of the Mathematical Sciences Department of the IBM T. J.Watson Research Center until he joined the EECS Department at Berkeley in 1987. He is a Fellow of the IEEE and a member of the National Academy of Engineering. Notable awards include: IEEE Emanuel R. Piore (2006); ACM Kanallakis (2006); European DAA Lifetime Achievement (2006); EDAC/CEDA Phil Kaufman (2007); D.O. Pederson best paper in Trans. CAD (2008); ACM/IEEE A. Richard Newton Technical Impact in EDA (2009); Iowa State University Distinguished Alumnus (2010); SRC Technical Excellence (2011); and the ACM/SIGDA Pioneering Achievement (2011). Prof. Brayton held the Buttner Chair and the Cadence Distinguished Professorship of Electrical Engineering and is currently a Professor in the Graduate School at Berkeley.