Improvements to Delay-driven LUT Mapping

Alessandro Tempia Calvino
EPFL, Switzerland

Giovanni De Micheli
EPFL, Switzerland

Alan Mishchenko
UC Berkeley, USA

Robert Brayton
UC Berkeley, USA

ABSTRACT

Ashenhurst-Curtis decomposition (ACD) is a known decomposition technique used, in particular, to map combinational logic into LUT structures when synthesizing hardware designs. However, available implementations of ACD suffer from excessive complexity and slow run time, which limits their applicability and scalability. This paper presents several simplifications leading to a fast and versatile technique of ACD suitable for delay optimization. We utilize this new formulation to enhance delay-driven LUT mapping by performing ACD on the fly. Compared to state-of-the-art technology mapping, experiments demonstrate an average delay improvement of 17.94%, with affordable run time. Additionally, our method improves heavily optimized LUT networks.

1 INTRODUCTION

Ashenhurst-Curtis decomposition (ACD) [2, 8], also known as Roth-Karp decomposition [22], is a powerful technique that finds a decomposition of 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. Noteworthy usages include functional decomposition for standard cell mapping [12] and LUT mapping [13], decomposition of multi-valued relations [20], and encoding of multi-valued networks [11].

More recently, ACD has been applied to map into *lookup table* (LUT) structures [21] as a way to mitigate structural bias and improve the quality of standard LUT mapping. Moreover, ACD has been used in post-mapping resynthesis [16], when logic cones composed of several LUTs are collapsed into single-output Boolean functions and re-expressed using fewer LUTs. These applications rely on the traditional formulation of ACD [2, 8, 22], breaking the input variables into two groups: the bound set (BS) and the free set (FS). For instance, Figure 1 shows an ACD of an 8-input function with a 6-variable BS and a 2-variable FS, resulting in three 6-input LUTs. Other known approaches to ACD [13] allow for a shared set (SS) when one or more LUTs in terms of the BS variables are single-variable functions (buffers). The larger the SS size, the fewer LUTs are required. In [13], maximizing the SS is implemented using *binary decision diagrams* (BDDs) [3], which makes it not applicable when Boolean functions are represented using truth tables.

Since ACD is applicable only to functions up to 16 inputs, state-of-the-art LUT mapping is performed through local substitutions applied to an initial graph representation, called *subject graph*. Generally, delay-optimal mapping w.r.t the subject graph is feasible in polynomial time [6], while area-optimal mapping is NP-hard [10]. However, the structure of the subject graph highly influences the result. This is known as *structural bias*. To mitigate structural bias, methods in the literature generate a set of structural choices available during mapping [4, 5, 14].

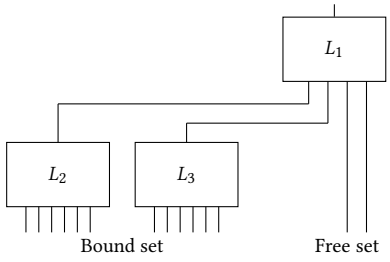

Figure 1: ACD of an 8-input Boolean function into 6-LUTs

This paper brings two main contributions. First, we revisit the known formulation of ACD with SS, aiming at making it computationally efficient in LUT mappers and post-mapping resynthesis engines. Our algorithm is truth-table-based and flexible in the number of FS, BS, and SS variables, which makes it suitable for delay optimization. Compared to previous implementations of LUT packing in ABC, it runs 3.5x faster when solving similar problems. Second, we use ACD for the delay optimization of LUT networks. The idea is to compute functional decompositions using the timing-critical variables in the FS and the rest of the variables in the BS and SS. This technique is more general than previous techniques based on cofactoring w.r.t. late arriving variables using Shannon expansion [17]. We integrate our method into an LUT mapper for delay optimization. To our knowledge, this is the first practical work that uses ACD for delay-driven LUT mapping.

In the experiments, we show that mapping with ACD can efficiently mitigate the structural bias and considerably reduce the delay at the cost of higher area and number of edges. First, we compare the default LUT mapper in ABC, the LUT-structure mapper in [21], and the proposed mapper with integrated ACD. We show that mapping with ACD notably outperforms the other mappers in delay by 9.52% 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 standard LUT mapper, by 17.94% with a slight area increase of 3.66%. Finally, we show that our method can improve the delay even for heavily optimized LUT networks, obtained using state-of-the-art algorithms in ABC until convergence.

2 BACKGROUND

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

2.1 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 $b = b_{l-1} \dots b_0$, i.e., a sequence of bits, of length $l = 2^k$. A bit $b_i \in \{0,1\}$ at position $0 \leq i < l$ is equal to the value taken by f under the input assignment $\vec{a} = (a_0, \dots, a_{k-1})$ where

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

In a classical representation, we 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 (two input columns) changes the truth table. For instance, the Boolean implication function $a_1 \rightarrow a_0$, encoded as 1011, changes to 1101 after the variable swap.

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 contain common variables. A set of functions is disjoint if their supports are pair-wise disjoint.

A *Boolean network* is modeled as a directed acyclic graph (DAG) with nodes represented by 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.

A *cut* C of a Boolean network is a pair $(n, \mathcal{K})$, where n is a 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 passing through v and not through another leaf. The *size* of a cut is the number of leaves. A cut is k-feasible if its size does not exceed k.

2.2 Ashenhurst-Curtis decomposition

Ashenhurst-Curtis decomposition (ACD) [2, 8, 22], of a single-output Boolean function f can be expressed as follows:

$$f(x_{bs}, x_{ss}, x_{fs}) = g(H(x_{bs}, x_{ss}), x_{ss}, x_{fs}),$$

where x_{bs} is the *bound set* (BS), x_{ss} is *shared set* (SS), and x_{fs} is the *free set* (FS). These sets are disjoint variable subsets, which together form the support of f. The function H may be multi-output with the number of outputs less than the BS size. The single-output functions in H are referred to as BS functions. The function g is referred to as the *composition function* and is typically chosen to fit into one k-input LUT. Figure 1 shows an ACD of an 8-input function into three 6-LUTs with a 6-variable BS, two BS functions (L_2, L_3), and a 2-variable FS.

2.3 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 *And-inverter graph* (AIG) [15] 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, which are computed using cut enumeration [7]. Since the substitutions are local, the structure of the subject graph influences, to a large extent, the structure of the mapped network. This is known as *structural bias*. To mitigate structural bias, most of the methods in the literature generate a set of decomposition choices available during LUT mapping [4, 5, 14].

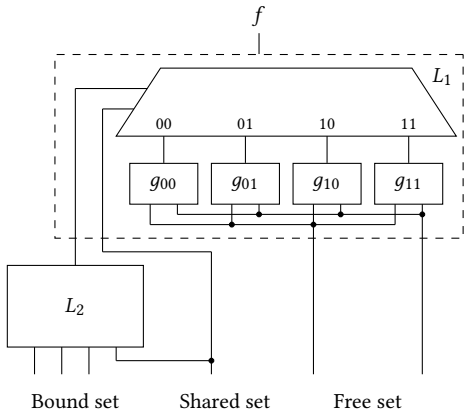

Figure 2: Our AC decomposition of a Boolean function

3 IMPROVEMENTS TO ACD

ACD is a powerful technique to describe an N-input Boolean function in terms of k-LUTs, with $N > k$. It is known that the quality of technology mapping strongly depends on the structure of the subject graph [4, 5]. Moreover, it has been shown that, for small functions up to 16 inputs, ACD typically performs better than structural mappers [13, 16]. Nevertheless, ACD is impractical for large functions and suffers from excessive complexity and slow run time.

This section discusses a fast and versatile truth-table-based implementation of ACD for single-output functions with support for a shared set. We propose several simplifications that make ACD practical within LUT mappers and resynthesis methods. Figure 2 shows the type of decomposition computed by our approach. The BS, SS, FS, and the number of BS functions used are determined during the decomposition. The composition function (L_1) is implemented as a multiplexer of cofactors with respect to BS functions. Functions dependent on the FS (g_{ij}) are the data inputs of the multiplexer found inside the composition function. BS functions and the shared set are instead the selection inputs. Our algorithm is divided into two main procedures. The first one enumerates different FSs and BSs to find a feasible decomposition. The second one finds an encoding of the BS functions while minimizing their support. Finally, we discuss how the SS can be maximized.

3.1 Computing free set and bound set

Given a truth table, our approach first enumerates different free sets to find feasible decompositions. This step is equivalent to enumerating different bound sets. Let N be the number of variables

in the support of a function to decompose. Let P be the number of variables to consider in the free set. 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 at least $(N-k)$ variables in the FS. For instance, when $N=8$ and $k=6$, there are $8 \cdot 7/2 = 28$ different 2-variable FS's.

For each FS, the truth table is transformed to have the FS variables as the least significant ones, compared to the BS variables. In this case, if the BDD of the function is available, the BS variables are ordered first (i.e., above the FS variables) after the transformation. When truth tables are used, the variable reordering is performed using a dedicated procedure, which swaps two variables. Note that 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. Every consecutive FS can be derived from a previous FS by one two-variable swap.

Each value assignment of the BS variables selects one (P)-input function in terms of the FS variables. From a truth table in this format, 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 listed next to each other.

Example 1: Let us consider the 6-variable function represented in hexadecimal format as a truth table f = 0x8804800184148111. Let us 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 $2^P = 2^2 = 4$ bits. There are $2^{(N-P)} = 16$ of them, 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 different FS functions, known as column multiplicity μ, does not exceed 2^M. The decomposition is feasible if $P + M \leq k$, such that the composition function can be implemented as a k-LUT. Much of the previous work on ACD tries to minimize M by selecting $M = \lceil \log_2(\mu) \rceil$ [16, 21]. In our approach, we don't minimize M since, in practical functions, some of the M functions can be buffers leading to a shared set, such as in Figure 2.

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 needs at least $\lceil \log_2(4) \rceil = 2$ BS functions. This decomposition is support-reducing and can be decomposed using 4-input LUTs. Using Figure 2, ACD assigns FS functions to $g_{i,j}$. Two BS functions of at most 4 inputs are instead used to select the correct FS function.

3.2 Encoding and support minimization

Once an FS with a feasible decomposition is found, the BS functions are extracted by solving an encoding problem. Let *i-sets* be the set of μ Boolean functions in terms of the BS variables that correspond to a one-hot encoding of the FS functions. Each i-set is 1 for each BS combination that leads to the FS function they represent.

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

It is crucial to find an encoding that minimizes the support of each BS function. Ideally, minimizing the support, each BS function would fit into a k-LUT, and the decomposition does not have to recur on BSs for further decomposition. Finding a feasible encoding is similar to solving constrained encoding problems [9, 24, 25]. Deriving this encoding involves two steps. The first one enumerates candidate BS functions. The second one solves an unate covering problem in which columns are candidate BS functions and rows are pairs of FS functions to be distinguished.

Candidate BS functions represent one variable of the encoding of the FSs. They are enumerated by combining i-sets. To leverage all the functional degrees of freedom, i-sets in a BS candidate can be either in the *ON-set*, *OFF-set*, or *don't-care* (DC) set. Since candidate BSs are select inputs of a multiplexer, BS candidates 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 the pairs of functions to be distinguished are referred to as *seed dichotomies* [25]. Don't-cares in BS candidates are essential to minimize the support, which translates into fewer edges in the LUTs.

Example 4: Continuing Example 3, let us consider the candidate bound set h that has {0x8, 0x1} in the ON-set and {0x4} in the OFF-set. Its function in 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. In this case, function h distinguishes the 0x8 from 0x4 and 0x1 from 0x4.

Candidate bound sets are enumerated by assigning each i-set to be in the ON-set, OFF-set, or DC-set. A total of 3^μ BS candidates are enumerated. Nonetheless, some BS candidates are interchangeable, i.e., one candidate can be obtained by swapping the ON-set and the OFF-set of another BS candidate. Our enumeration removes these symmetries by fixing one i-set to be only in the ON-set or DC-set, enumerating only $2 \cdot 3^{\mu-1}$ BS candidates. Moreover, candidates that do not distinguish at least one pair of FS functions are removed. As a special case, if μ is a power of 2, the enumeration does not use the DC-set and splits the FS functions to be equally distributed between ON-set and OFF-set, i.e., each BS candidate must distinguish half of the FS against the other half. Hence, the number of possible BS candidates is reduced to $\binom{M}{M/2}/2$. This is a necessary condition for a solution that distinguishes all pairs of functions.

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 with a special procedure that considers don't cares. Then, the covering table is constructed by having all the pairs of FS functions to be distinguished (seed dichotomies) as rows. The candidate BS functions are columns. A row-column entry (i, j) is 1 if the BS candidate of column j distinguishes the pair i. Then, a solution that minimizes the support is computed by solving a minimum-cost covering problem. The solution must cover all the rows while minimizing the cost. If μ is less than 5, the minimum cover is computed exactly using branch and bound. For other values of μ, we use greedy covering followed by local search.

Example 5: Figure 3 shows a covering table reflecting the examples in this section. Each column in the table is a candidate BS function shown as a truth table in hexadecimal format on 4 variables. Each BS candidate has a cost based on the number of

	4 *c9af*	3 1177	3 2727	3 *d8d8*	3 *ee*88	4 3650
{{0x8}, {0x0}}	1	0	1	1	0	1
{{0x8}, {0x4}}	1	1	0	0	1	1
{{0x8}, {0x1}}	0	1	1	1	1	0
{{0x0}, {0x4}}	0	1	1	1	1	0
{{0x0}, {0x1}}	1	1	0	0	1	1
{{0x4}, {0x1}}	1	0	1	1	0	1

Figure 3: Covering table to solve the encoding problem

variables on its support. 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, for instance taking the second and third columns, has cost 6 and results in two BS functions with support of 3 variables. Note that Figure 3 also shows redundant columns that are not enumerated by our algorithm.

Given a solution, an encoding of the FS functions is obtained by assigning a code $T = t_1 \dots t_M$ in which each variable t_i corresponds to a selected BS_i candidate. Variable t_i takes either value 1, 0, or − if the FS function is in the ON-set, OFF-set, or DC-set of BS_i, respectively.

Example 6: Continuing Example 5, a minimum cover involves BS_1 = 0x1177, by taking 0x4 and 0x1 in the ON-set, and BS_2 = 0x2727 by taking 0x0 and 0x1 in the ON-set. Both bound sets depend only on 3 variables. Given, the BS functions, the encoding codes for the FS functions, which are assigned to $g_{t_1 t_2}$ of Figure 2, are T_{0x8} = 00, T_{0x0} = 01, T_{0x4} = 10, and T_{0x1} = 11. Finally, the composition function is computed using the FS and its encoding, obtaining function 0x1048, in hexadecimal format. Consequently, the function has been successfully decomposed using three 4-LUTs.

3.3 Maximizing the shared set

The number of LUTs required to implement the BS functions can be minimized using the shared set. To check whether a decomposition with $L \in [0, M)$ single-variable functions (or buffers) and $M - L$ non-buffer BS functions exists, the proposed method enumerates subsets of L out of $N - P$ variables in each BS. For each subset, the method checks if the number of unique FS functions in each cofactor w.r.t. L variables does not exceed 2^{M-L}. If this is the case, a decomposition with L buffers exists. This check is performed for all $\binom{N-P}{L}$ subsets and for each considered FS.

4 TECHNOLOGY MAPPING WITH ACD

In this section, we leverage Ashenhurst-Curtis decomposition (ACD) to improve the delay of a LUT network. ACD can be used in two ways: 1) as part of LUT mapping, or 2) as a post-mapping resynthesis method to pack logic and decrease the delay. In this work, we focus on the former usage since it has more flexibility and optimization opportunities at the cost of a higher run time.

First, we present delay-oriented functional decomposition using the methods described in Section 3. Then, we describe how ACD is employed during technology mapping.

4.1 Delay-oriented ACD

Let us consider a k-LUT network with a node n 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 delay of a leaf in C. We use ACD to find an implementation that realizes the function of cut C with delay $D + 1$, assuming a unit-delay model, when $|C| > k$. Specifically, we use the timing-critical leaves of C in the FS, and other non-critical ones in the BS or SS. This transformation can reduce the worst delay of a LUT network when applied on a critical path. Note that this decomposition is different from cofactoring w.r.t. late arriving variables based on Shannon expansion [17] because it performs a more general decomposition of the target function. Compared to [17], which connects late-arriving signals at the select lines of a top MUX, this approach uses critical signals as data inputs of a top MUX. Hence, this method is orthogonal to previous state-of-the-art delay optimization techniques for LUT networks. In this work, we focus on ACD decompositions involving only two levels of LUTs.

The ACD-based transformation is computed in two steps. First, our method verifies the existence of a delay-minimizing decomposition. Second, if a decomposition exists, our algorithm solves the encoding problem while minimizing the support and the shared set to return an implementation. Algorithm 1 shows the procedure *evaluate* to check the existence of an AC decomposition. The algorithm receives the function represented as a truth table tt belonging to a large cut of size N with more than k leaves. Set S contains a list of timing-critical variables with delay D. Then, the truth table is transformed to have the late arriving variables as the least significant ones, since they must be the in FS (at line 3).

The proposed approach limits $|BS \cup SS| \leq k$ to ensure a two-level decomposition. Hence, the number of variables in the FS must be at least $N - k$, and $|S|$ to include all the delay-critical variables (at line 6). Our implementation currently supports up to 5 FS variables. For each size P_i of the FS, the smallest column multiplicity value is computed using the method described in Section 3.1. In this case, since delay-critical variables are always part of the FS, $\binom{N}{P_i - |S|}$ different combinations are enumerated (at line 7). If the smallest multiplicity found can be implemented using at most $k - P_i$ BS functions, a delay-minimizing ACD exists. In that case, Algorithm 1 assigns the propagation delay to each variable. Variables in the FS have a delay of 1. Others have a delay of 2 (at line 12).

After the procedure *evaluate*, another one computes the actual decomposition using the methods described in Sections 3.2 and 3.3.

4.2 LUT mapping with ACD

The methods described in Section 4.1 have been integrated into an LUT mapper. State-of-the-art technology mapping typically performs one delay minimization followed by several iterations to recover area. Typically, enumerated cuts are k-feasible, i.e., any cut abstracts a k-LUT. In our implementation, we use ACD decomposition on the fly during delay minimization for large cuts of size $k < l \leq 10$, where l is provided by the user. The following area recovery iterations can either use ACD on large cuts or standard k-feasible cuts. Only the ACD evaluation (Algorithm 1) is used during the mapping iterations. The actual decomposition is computed when the LUT network is returned for the non-k-feasible cuts. In

Algorithm 1: ACD evaluation

```
Input : Truth table tt, LUT size k, Late vars set S
Output: Propagation delay
reorder_vars(tt, S);
μ_best ← ∞;
P_best ← 0;
for P_i ← max(num_vars(tt) − k, |S|) to 5 do
    μ ← compute_smallest_multiplicity(tt, P_i, |S|);
    if μ ≤ 2^(k−P_i) and μ ≤ μ_best then
        μ_best ← μ;
        P_best ← P_i;
if μ_best = ∞ then
    return infinite_propagation_delay();
return compute_propagation_delay(tt, P_best);
```

the integration, if an ACD of a cut C and set S is not feasible, a decomposition with $S = \emptyset$ is attempted. In fact, an implementation with delay $D + 2$ may still be better than any structural k-feasible cut.

5 EXPERIMENTS

This section presents experimental evaluation of the proposed LUT mapping with ACD. The proposed algorithms have been implemented in *ABC*[1]. For our experiments, we use the EPFL combinational benchmark suite [1] containing several circuits provided as *and-inverter graphs* (AIGs). The baseline has been obtained using the commands and scripts "`dfraig; resyn; resyn2; resyn2rs; if -y -K 6; resyn2rs;`" in ABC, which perform a strong AIG-based depth and size optimization. In particular, it combines SAT sweeping [19], scripts for delay-oriented AIG optimization [15], and lazy man logic synthesis [26], which is the most aggressive depth minimization command in ABC. The experiments have been conducted on an Intel i5 quad-core 2GHz on MacOS. All the results were verified using the combinational equivalent checker in ABC. We extended the LUT mapper *if* in ABC to perform ACD according to Section 4. In this implementation, ACD is used only in the delay-oriented iteration. The following commands are used in the experiments:

- `dch (-f)`: computes structural choices to mitigate the structural bias [4], where `-f` stands for "fast";
- `if -K 6`: performs technology mapping into 6-LUTs using 6-feasible cuts;
- `if -s -S 66 -K 8`: performs technology mapping using 8-feasible cuts and packs logic into a structure composed of two 6-LUTs using a variant of ACD from [21];
- `if -z -Z 6 -K 8`: performs technology mapping into 6-LUTs using our delay-oriented implementation of ACD on 8-feasible cuts;
- `st`: derives an AIG from a LUT network.

5.1 Delay-driven LUT mapping

Table 2 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 -Z 6 -K 8`".

Mapping into LUT structures 66 composed of two 6-LUTs, which is based on a limited version of ACD, reduces depth by 1.56% on average, at the cost of increasing the number of LUTs and edges by 1.96% and 1.93%, respectively. The proposed LUT mapping with ACD improves the depth of the LUT network by 9.52% on average while increasing the number of LUTs and edges by 9.52% and 9.73%, respectively. Note that most of the improvement is concentrated in the first 10 benchmarks since most of the others are already close to their optimal depth. Practically, part of the area increase can be reduced by area-recovery methods [18, 23], using delay relaxation or by an additional mapping step applied after ACD. The rightmost strategy performs the latter option. Both the LUT count and edge count are reduced considerably. Also, the depth decreases. Specifically, the result after ACD is used as a choice to improve the next round of technology mapping, because choices extracted from mapping with ACD are more structurally suited to delay-oriented mapping, compared to the original AIG. Note that a second mapping round does not provide practical benefits if applied to the default LUT mapper (leftmost column) since the network after deriving the AIG is structurally similar to the baseline. Regarding the run time, mapping with ACD is much faster than mapping into LUT structures while being more general. Moreover, compared to default structural LUT mapping, some run time overhead comes from computing the truth tables of the cuts needed for ACD.

Table 1: ACD applied to highly optimized LUT networks.

Benchmark	Aggressive opt.		ACD post-opt.	
	LUTs	Depth	LUTs	Depth
adder	501	7	541	6
bar	512	4	512	4
div	16086	257	22513	222
hyp	74550	715	98447	599
log2	8778	60	10548	53
max	1180	12	1341	11
multiplier	6101	32	6864	27
sin	1618	33	1935	28
sqrt	12780	312	19266	251
square	4057	15	4302	11

5.2 Improving heavily optimized designs

This experiment shows that mapping using ACD can improve heavily optimized LUT networks. We obtain initial mappings by optimizing the LUT networks from the leftmost strategy of Table 2 until convergence. Specifically, we use command `speedup`, which performs cofactoring w.r.t. the latest arriving inputs of LUTs [17], in addition to a combination of the non-ACD-based commands introduced before. Then, we perform one iteration of delay-oriented LUT mapping with ACD with the script "`st; dch -f; if -z -Z`

[1]Available at: https://github.com/berkeley-abc/abc

Table 2: Comparison of delay-driven LUT mapping, LUT mapping to structures, and LUT mapping using ACD.

Benchmark	dch; if -K 6				dch; if -s -S 66 -K 8				dch; if -z -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	273	1024	51	0.28	399	1294	43	0.53	329	1319	37	0.26	317	1320	20	0.37
bar	512	2688	4	0.40	512	2688	4	1.37	512	2688	4	0.53	512	2688	4	0.71
div	8618	32394	406	8.53	9107	32753	397	22.64	12256	47091	328	8.78	9535	39864	280	20.25
hyp	44508	198047	4194	4.90	52619	220926	4156	68.06	52739	229022	2835	17.09	53273	234293	1401	26.04
log2	8296	37508	67	10.68	8619	36678	67	33.99	8572	38823	62	13.33	8510	38535	59	35.42
max	820	3407	34	0.46	840	3524	33	2.02	1149	4695	25	0.58	859	3760	17	0.85
multiplier	6092	28085	53	4.79	6023	25341	53	18.92	6235	28795	44	6.62	6381	28882	37	12.52
sin	1481	6884	35	1.53	1603	6770	33	7.56	1941	8688	32	2.10	1869	8371	32	3.43
sqrt	5126	22895	990	5.22	5068	22554	997	18.41	6595	29963	692	6.27	6038	25087	500	10.79
square	4122	17319	23	2.30	4164	17443	22	8.20	4128	17929	18	2.98	4119	18285	14	6.20
arbiter	1833	8982	6	1.80	1879	8681	6	3.53	1850	8992	6	2.08	2054	8814	6	3.85
cavlc	137	707	4	0.14	104	458	4	0.79	137	707	4	0.17	123	655	4	0.24
ctrl	30	133	2	0.07	29	124	2	0.14	30	133	2	0.07	29	126	2	0.08
dec	287	684	2	0.10	287	1388	2	0.19	287	684	2	0.10	284	816	2	0.12
i2c	312	1360	3	0.19	307	1277	3	0.63	316	1376	3	0.22	301	1358	3	0.30
int2float	52	258	3	0.09	46	190	3	0.24	52	258	3	0.09	50	251	3	0.11
mem_ctrl	11037	48812	18	12.02	10557	44588	18	44.42	11051	48825	18	14.24	10200	45235	16	24.72
priority	178	725	6	0.14	182	715	6	0.28	180	718	6	0.14	167	711	6	0.18
router	89	285	4	0.10	88	274	4	0.19	89	285	4	0.10	87	273	4	0.12
voter	1838	8596	13	2.47	1784	8538	13	5.83	1839	8591	13	2.80	1776	8395	13	5.58
Improvement					-1.96%	-1.93%	1.56%		-9.52%	-9.73%	9.52%		-3.66%	-5.45%	17.94%	
Total				56.21				237.94				78.49				151.88

`6 -K 10`", using cuts of size 10. The results are shown in Table 1. Our method improves the depth of 9 out of 10 benchmarks.

6 CONCLUSION

This work revisits 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 implemented and integrated the proposed formulation into a delay-driven LUT mapper. The experiments have shown that LUT mapping with ACD can improve the average delay by 17.94%, compared to traditional structural LUT mapping with choices. Moreover, the proposed approach leads to delay reductions even when applied to heavily optimized LUT networks.

REFERENCES

[1] L. Amarù, P.-E. Gaillardon, and G. De Micheli. 2015. The EPFL Combinational Benchmark Suite. In *Proc. IWLS*.

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

[3] R. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Computers* C-35, 8 (1986), 677–691.

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

[5] G. Chen and J. Cong. 2001. Simultaneous Logic Decomposition with Technology Mapping in FPGA Designs. In *Proc. FPGA*. 48–55.

[6] J. Cong and Y. Ding. 1994. FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *Trans. CAD* 13, 1 (1994), 1–12.

[7] J. Cong, C. Wu, and Y. Ding. 1999. Cut Ranking and Pruning: Enabling a General and Efficient FPGA Mapping Solution. In *Proc. FPGA*.

[8] J. P. Curtis. 1962. A New Approach to the Design of Switching Circuits. *Proc. Int. Symp. Theory Switch.*

[9] G. De Micheli, R.K. Brayton, and A. Sangiovanni-Vincentelli. 1985. Optimal State Assignment for Finite State Machines. *Trans. CAD* 4, 3 (1985), 269–285.

[10] A. H. Farrahi and M. Sarrafzadeh. 1994. Complexity of the lookup-table minimization problem for FPGA technology mapping. *IEEE Trans. CAD* (1994).

[11] J.-H. Jiang, Y. Jiang, and R. K. Brayton. 2001. An implicit method for multi-valued network encoding. In *Proc. IWLS*. 127–131.

[12] V. N. Kravets and K. A. Sakallah. 2001. *Constructive Multi-Level Synthesis by Way of Functional Properties.* Ph. D. Dissertation.

[13] C. Legl, B. Wurth, and K. Eckl. 1998. Computing support-minimal subfunctions during functional decomposition. *Trans. VLSI* 6, 3 (1998), 354–363.

[14] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness. 1997. Logic decomposition during technology mapping. *Trans. CAD* (1997).

[15] A. Mishchenko and R. Brayton. 2006. Scalable Logic Synthesis using a Simple Circuit Structure. In *Proc. IWLS*.

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

[17] A. Mishchenko, R. Brayton, and S. Jang. 2010. Global Delay Optimization Using Structural Choices. In *Proc. FPGA*. 181–184.

[18] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang. 2011. Scalable Don't-Care-Based Logic Optimization and Resynthesis. *ACM Trans. Reconfigurable Technol. Syst.* 4, 4 (2011).

[19] A. Mishchenko, S. Chatterjee, and R. Brayton. 2005. *FRAIGs: A unifying representation for logic synthesis and verification.* Technical Report. EECS Dep., UC Berkeley.

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

[21] S. Ray, A. Mishchenko, N. Een, R. Brayton, S. Jang, and C. Chen. 2012. Mapping into LUT structures. In *Proc. DATE*.

[22] J. P. Roth and R. M. Karp. 1962. Minimization Over Boolean Graphs. *IBM Journal of Research and Development* 6, 2 (1962), 227–238.

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

[24] T. Villa and A. Sangiovanni-Vincentelli. 1990. NOVA: state assignment of finite state machines for optimal two-level logic implementation. *Trans. CAD* 9, 9 (1990), 905–924.

[25] S. Yang and M.J. Ciesielski. 1991. Optimum and suboptimum algorithms for input encoding and its relationship to logic minimization. *Trans. CAD* 10, 1 (1991), 4–12.

[26] W. Yang, L. Wang, and A. Mishchenko. 2012. Lazy Man's Logic Synthesis. In *Proc. ICCAD*. 597–604.