LUT-Based Optimization For ASIC Design Flow

Luca Amarú*, Vinicius Possani*, Eleonora Testa*, Felipe Marranghello*, Christopher Casares*,
Jiong Luo*, Patrick Vuillod*, Alan Mishchenko‡, Giovanni De Micheli†

*Synopsys Inc., Design Group, Sunnyvale, California, USA
†Integrated Systems Laboratory, EPFL, Lausanne, Switzerland
‡Department of EECS, University of California, Berkeley, USA

***Abstract*— Look-up Table (LUT) mapping and optimization is an important step in *Field Programmable Gate Arrays* (FPGAs) design. The effectiveness of LUT synthesis improved dramatically in the last decades, thanks to optimization and mapping innovations naturally tailored for FPGAs. In this paper, we develop a new LUT-based optimization flow that is tailored for the synthesis of *Application-Specific Integrated Circuits* (ASICs) rather than FPGAs. We enhance LUT mapping to consider the literal/AIG cost of LUT nodes. We extend traditional Boolean methods to simplify and re-shape LUT-networks, targeting the best AIG/mapped-network implementation, after decomposition. Intuitively, literal-driven LUT packing behaves as a powerful fanin-bound node elimination, unveiling higher-order Boolean simplification opportunities. We embed our proposed LUT-based optimization flow, area oriented, in a commercial synthesis tool. Using our methodology, we improve 12 of the best area results in the EPFL synthesis competition. Employed in a commercial EDA flow for ASICs, our LUT optimization reduces area by 1.80%, total negative slack by 0.39%, and switching power by 1.72%, after physical implementation, at 5% runtime cost.**

I. Introduction

Look-Up Table (LUT) synthesis is a key step in *Field Programmable Gate Arrays* (FPGAs) design. In the last decades, innovations in mapping and minimization of LUT networks has improved FPGA implementations significantly [1]–[3]. While naturally tailored for FPGAs, LUT synthesis often provides intrinsic complexity reduction in the representation of logic networks. In this paper, we exploit LUT-mapping and LUT-minimization technologies applied to standard-cell based *Application-Specific Integrated Circuits* (ASICs) design, in addition to their traditional use in FPGA design.

The rationale behind this work is to strike a better balance between functionality and structure in logic optimization for ASICs. Functionality is dominant when large nodes are used, while the structure takes over when the network is implemented using small nodes. Large nodes reveal more don't-care-based optimizations, while small nodes provide more cut points for network restructuring. It is an open problem how to best harmonize functionality with structure. With our innovations, we offer an automated and efficient solution to this problem, tailored to ASIC design, pushing the boundaries of the *Quality of Results* (QoR) achievable by logic synthesis.

This paper presents a novel, LUT-based, Boolean optimization flow embedded in a commercial synthesis tool for ASICs. We revisit and expand on several notions of LUT mapping and adapt Boolean techniques to exploit the compression power of LUT nodes. Our optimized LUT networks can be naturally decomposed into smaller and faster AIGs or mapped networks. More specifically, the contributions of this paper are:

- Enhanced LUT mapping, considering factored literals or AIG cost of each LUT node.
- Specialized Boolean methods for simplifying LUT networks, targeting the best AIG or cell-mapped implementation, after decomposition.
- Complete ASIC synthesis embedding our new LUT optimization flow, including traditional Boolean resynthesis and synthesis with structural choices [1].

We evaluate our techniques on both academic and industrial benchmarks. By using the proposed LUT optimization flow directly on the EPFL combinational benchmarks [4], we improve 12 of the best area results. We embed the proposed LUT optimization flow in a commercial EDA tool. After physical implementation, our flow reduces the area by 1.80%, total negative slack by 0.39%, and switching power by 1.72%, at 5% runtime cost.

The remainder of this paper is organized as follows. Section II gives some background on LUT mapping and Boolean optimization. Section III revisits LUT mapping algorithm, with an ASIC implementation objective. Section IV proposes effective Boolean optimizations that natively exploits LUT compaction. Section V details our new LUT-opto synthesis flow, stemming from the methodologies in Sections III and IV. Section VI shows experimental results over industrial benchmarks and compares the results to state-of-the-art solutions. Section VII concludes the paper.

II. Background

This section provides background on LUT mapping and Boolean methods for synthesis.

A. LUT Mapping

In LUT mapping, a logic network is covered with k-bounded lookup tables (k-LUTs) where each k-LUT can represent any function of k variables. Several methods for LUT mapping have been proposed in the last decades. The FlowMap algorithm, proposed in [2], obtains a minimum depth k-LUT cover. The previous work [5] partitions the initial graph into a forest of trees, maps each tree individually, and combines the mapping for each tree to find a cover for the initial network. The methods in [3], [6], [7] instead collapse the network into nodes with more than k variables and decompose them to obtain a k-LUT mapping. In [8], a cut-merging technique to enumerate k-cuts in the context of re-timing is presented, serving as basis for future k-cut enumeration approaches. In [9], runtime is improved by using only a small subset

978-1-6654-3274-0/21/$31.00 ©2021 IEEE

of good k-feasible cuts for each node, known as priority cuts. Improvements including speed up of cut computation and area recovery are presented in [1]. Several works aim at reducing the structural bias of technology mapping, that is the dependence of the mapped network on the initial structure [1], [10]. *Functionally Reduced AIGs* (FRAIGs) use a combination of simulation and combinational equivalence checking (SAT) to find equivalent (up to complement) AND nodes [10]. These AND nodes are merged into choice nodes. Intuitively, a choice node can be seen as a vertex that encodes different implementations of a function (up to complementation). Alternatively, a synthesis flow with structural choices, also called lossless synthesis flow, generates choice nodes by storing nodes generated in intermediate steps of the synthesis process [1]. In contrast, a traditional technology mapper only sees the nodes in the network resulting from the optimizations process. In [11], a framework to combine logic transformations with technology mapping is proposed. At each step the mapper evaluates the impact of a logic transformation on the mapped circuit.

B. Boolean Methods

Approaches to logic network optimization are divided into algebraic methods and Boolean methods [12]. Compared to algebraic methods, Boolean methods are based on Boolean transformations that consider the true nature of logic functions and improve logic networks through the freedom provided by don't care conditions [13]. *Observability don't care* (ODC) and *controllability don't care* (CDC) are widely used in synthesis with don't cares. In this work, we use the terms ODC/CDC and *permissible functions* interchangeably to express the following: if the function at node n can be changed into another function without changing the behaviour of the primary outputs, the new function is called a permissible function. The set of all permissible functions for a node n is called its *Maximum Set of Permissible Functions* (MSPF, [14]), and can be used to enhance Boolean transformations such as resubstitution, refactoring, rewriting, etc. Due to the use of don't cares and Boolean identities, Boolean methods achieve better QoR but usually have higher computational cost. Consequently, many recent works have been focused on improving their scalability [15], [16]. As of today, different reasoning engines can be used for detecting permissible functions and don't care conditions, and the right choice of such engine can improve the scalability of the Boolean methods. Examples of such engines – as used in Section IV – are truth tables, BDDs, and SAT. Details on the reasoning engines and their use in Boolean methods can be found in [15], [16] and [17], respectively. We also refer the interested reader to [12], [18] for a more exhaustive review of Boolean methods and transformations.

III. Revisiting LUT Mapping

In this work, we are interested in area-oriented LUT mapping. We make LUT mapping operating as a basic restructuring transform, such as a fanin-bounded eliminate, reshaping a generic Boolean network N in a more convenient state for ASIC synthesis. We present here our revisited LUT mapper, depicted in Algorithm 1. Initially, the mapper runs a preprocessing by sweeping the network N to (i) clean up single input gates (buffers and inverters), (ii) perform constant propagation and (iii) run a quick pass of a tree-mapping based on the chortle algorithm in [5]. After the preprocessing, the k-cut enumeration creates a set of cuts for each internal node using a cut-merging technique based on [8]. If the network comprises choices [1], the mapper can process them during the k-cut enumeration step to increase the mapping opportunities.

Algorithm 1 Top-Level LUT Mapping Algorithm

Input: Network N, LUT size k, lut_ratio r, backtrack b
Output: k-bounded network N' of LUTs

1: preprocess_network(N);
2: /* enumerate cutsets and process choices if available */
3: C = find_cutsets(N, k);
4: /* form the matches considering literal costing */
5: M = compute_matches(N, C, r);
6: /* form and solve the binate covering problem */
7: S = binate_cover(N, M, b);
8: /* re-form the network from the solution */
9: N' = collapse_network(N, S); return N';

TABLE I
Area-Oriented LUT Mapping Results Compared to ABC $\&if$.

	ABC $\&if$ k=3			Proposed k=3, r=1.4		
benchmark	LUTs	literals	levels	LUTs	literals	levels
adder	256	2,292	128	256	2,292	128
dec	298	620	3	304	**612**	3
i2c	745	2,339	12	754	**2,261**	12
max	1,290	4,682	158	**1,248**	**4,408**	189
mem_ctrl	25,361	88,038	78	**23,667**	**78,851**	94
log2	27,816	93,711	398	**25,447**	**88,473**	374

Typically, a conventional LUT-based mapper does not consider the literal/AIG count of the internal logic of a LUT. However, since we apply the LUT mapper to re-shape the network for ASIC synthesis, we need to estimate the internal LUT area to distinguish the cost of selecting a smaller or a larger cut during the mapping process. Therefore, the proposed mapper computes possible matches of cuts into LUTs by considering the implicit literal/AIG count of k-cuts into LUTs. When using small LUTs, e.g., 3-LUTs, it is quite efficient to use an lut_ratio parameter to determine the ratio among the area cost of different cut sizes. The intuition is that a 2-LUT maps often to a single AIG node, unless it represents a XOR, while a 3-LUT has higher complexity. Using a lut_ratio of 1.4 for $k = 3$[1] gives better LUT mappings that translate in smaller ASIC implementations. When considering instead higher k values, it is necessary to estimate the real area of each candidate LUT cone of logic. This is achieved by running a quick cube and kernel extraction on a duplicated cone of logic and by saving the resulting literal/AIG count as cost of the LUT. While such accurate costing has runtime impact, the proposed mapper reformulation reduces the number of LUTs while selecting k-cuts/matches with better literal count.

The next mapping step (line 7 in Algorithm 1) is to select a subset of all possible k-cuts/matches that cover the network, while minimizing the overall area cost. This DAG-covering is formulated as a binate covering problem by forming a single matrix for all the matches and their associated covered nodes. The binate covering is guided by the area cost of each match and the solving process is bounded by a given backtracking limit [19]. Branch-and-bound is a powerful tool for this kind

[1]In other words, $lut\_ratio = 1.4$ with $k = 3$ means that a 3-LUT has 1.4× the area cost of a 2-LUT.

of intractable problems since it can provide a high quality solution within a given computational effort limit. Finally, the mapper delivers a k-bounded network N' of LUTs by locally collapsing nodes into their fanins up to reach the k-cuts in S, selected during the covering process.

Table I presents a comparison between the proposed mapper and the ABC mapper $\&if$ in an area-oriented scenario. We ran the proposed mapper with $k = 3$, $r = 1.4$, and ABC mapper as follows *"r circuit.blif; st; &get; &if -K 3 -a; &put; w mapped.blif; q"*. Our mapper has demonstrated strength for reducing LUT and literal count, and also acceptable runtime for a high-quality mapper, 1.6 seconds on average for Table I. The blif files used as input to the mappers are available at [20].

IV. Boolean Methods for LUT Simplification

This section revisits Boolean methods with application to LUT optimization flow targeting ASICs.

A. LUT Complexity Minimization

Reducing the intrinsic complexity of LUT nodes in a LUT network is an important step for improving the ASIC implementation cost, after AIG decomposition and standard cell mapping. Reducing the LUT node complexity can be seen as reducing the NAND2/AIG cost of the same node. Effective ways to achieve this goal are to (i) reduce the cardinality/support-size of an LUT node, (ii) run 2-level *Sum Of Products* (SOP) minimization on the LUT node, including don't cares.

1) SOP simplify with don't cares: SOP simplification employs traditional 2-level minimization algorithms [21], [22]. In the context of LUT nodes, SOP simplification can be made very powerful because a tight bound on the support size is set by construction, thus higher effort minimizations can be run without incurring in intractable runtime. In practice, exact 2-level minimization methods [22] can be used with 3,4,5 fanin bound with quick runtime. However, just running simplification on such small SOP nodes does not usually lead to consistent reductions. To unlock more simplifications, we consider don't cares. More specifically, LUT networks offer the unique opportunity to add don't cares to SOP simplification in a bounded way. The CDC of an LUT node can be added by just considering one-level fanin of LUT nodes, and the ODC by considering one-level fanout of LUT nodes, up to a maximum fanout value F. The CDC complexity is proportional to k^2, where k is the LUT size, while the ODC complexity is proportional to $F \cdot k$. All these values are constants in practice, decided beforehand. Other enhancements to SOP simplification for LUTs regards the acceptance criterion. Rather than accepting based on literal count decrease, we can improve the cost to consider factored literals, by running quick extraction on the simplified SOP. This cost has better correlation to a final ASIC implementation. Lastly, both phases can be tried during LUT node SOP simplification with don't cares, and accept the best polarity leading to the smallest factored literals cost, including the output inverter if necessary.

2) Support reduction: When considering a final ASIC implementation, it is desirable to reduce the support size of each LUT node. For example, by reducing a 3-LUT into a 2-LUT, if possible, before decomposing and mapping onto standard cells. Indeed, LUT nodes could have redundant inputs when considering the global network functionality. We reduce the support of each LUT node by computing the functional support using either BDD or SAT based methods [7], [16], [17]. The functional support can be computed with respect to a frontier, i.e., a set of internal variables in the network, or with respect to the primary inputs of the network. When the functional support size is smaller than the LUT size of the node, the old SOP is replaced with a new ISOP computed based on the functional support. Either BDD or SAT based methods can be used to compute ISOPs efficiently in this context. SAT based methods are preferable when the frontier employed for computation is deep, i.e., going to the primary inputs.

B. LUT-Enhanced Boolean Resubstitution

Boolean resubstitution [15], [17] can be remarkably effective when run on an LUT network. Resubstituting and saving one large LUT node, or replacing a k-LUT with a (k-j)-LUT, reduces the complexity of the network more significantly than just resubstituting one AIG node. Compressing the network into LUTs increases the visibility of Boolean resubstitution so that higher order opportunities can be found. For LUT-optimization for ASICs purposes, it is important to make Boolean resubstitution cost-accurate and capable of the strongest resubstitution possible, in terms of globality and n-arity[2] In this work, we present a special Boolean resubstitution algorithm tailored for maximum visibility. Note that this Boolean optimization is high effort and capable of grinding down area significantly. Also, several runtime bailouts are in place to contain its runtime, as typically done for the most powerful Boolean transformations. The algorithm works as follows. First, the LUT nodes are ordered based on maximum savings at each node. Here, the maximum saving is the factored literal cost of the *Maximum Fanout-Free Cone* (MFFC) of the node. Then, each LUT is processed to find other nodes that are *connectable*, up to complementation, to realize the function of the target LUT via an OR operator. The function of the target LUT natively embeds MSPF flexibilities by construction, to find the largest set of *connectable* nodes. The MSPF and *connectability* computations are naturally performed with BDD operations, when BDDs can be built efficiently. SAT formulation and solving for the same computations is preferable in the large scale scenario. In the small case scenario (up to 15 inputs), truth tables can be used.

Once a set of connectable new fanins is found, a branch and bound algorithm is employed to determine the minimum irredundant subset of new fanins that, once connected via an OR operator, with complementation as needed, can implement the original LUT functionality under MSPF. Note that such new solution may not always exist, in which case the minimum irredundant subset would be empty. The value k from the LUT mapping is useful to prune the search space of the branch and bound problem. Complete functionality needs to be checked during the branch and bound problem, so either BDD, SAT, or truth table packages are called as needed. If a solution is found, the fanin can be re-arranged to further improve the solution,

[2]Here, resubstitution n-arity refers to the maximum support size of the new nodes introduced by resubstitution.

e.g., reduce number of factored literals. Once committing the resubstitution operation and disconnecting the old fanins, it is important to refresh any global function data structure as MSPF information may need updating.

While this Boolean resubstitution methodology may appear runtime intensive, it scales well for many small and medium size designs. For large designs, partitioning is the preferred strategy to use this powerful resubstitution without incurring an intractable runtime. Also, several deterministic guards on runtime are in place to bailout when branch and bound becomes longer than the intended maximum budget for the resubstitution optimization. Altogether, this makes LUT-enhanced Boolean resubstitution a very powerful, yet runtime affordable, optimization technique.

C. Boolean Rewiring Revisited for LUTs

Boolean rewiring [23] is a popular synthesis technique. It consists of a sequence of additions and removals of redundant wires, with the goal of reducing the (factored) literal cost of the network. Inserting a new redundant wire to an irredundant circuit can make two or more wires to be redundant, under certain conditions, thus leading to new simplification opportunities. Inserting a new redundant wire means introducing a new gate, or swapping to a wider gate, capable of receiving the extra connection. ATPG engines are historically quite efficient to detect such redundancies [24]. Boolean rewiring is classically a greedy technique, where single rewiring operations are immediately accepted when positive gain is observed. Zero gain moves are useful as well to escape local minima.

In our LUT optimization framework, Boolean rewiring is extended to operate on LUTs and introduce 2-input LUT nodes, *i.e.,* AND/OR, for the new redundant wires. Boolean rewiring is a multi-node optimization technique, for this reason it provides diversity w.r.t. the previous methods.

D. From LUT to AIG and Mapped Networks

The last step in LUT optimization for ASICs consists of decomposing the LUT nodes into a factorized form and, in turn, decomposing the network into an AIG or a mapped network. To obtain a factorized form of the SOP of each node, the logic function is extracted from the LUT and decomposed into primitive gates. The decomposition is achieved by recursively applying kernel extraction. For each node f in the network, the recursive kernel decomposition works as follows. If a non-trivial divisor g is found, f is rewritten using divisor g as literal, and the algorithm is recursively applied on both f and g. If a divisor is not found, the procedure returns the node itself. The output of the algorithm is an array of all nodes involved in the factorized form of f, which can be used to obtain an AIG or a mapped network. A final pass of light AIG-engine [16] further improves QoR.

V. LUT Optimization Flow for ASICs

This section presents a new LUT optimization framework targeting ASICs. It combines the new technologies detailed in Sections III and IV in an intelligent gradient-based engine. For the sake of brevity, this section will focus on AIG as target output network. However, mapped networks are a natural extension of this methodology.

Algorithm 2 LUT Engine Optimization Flow

Input: AIG N, LUT size k, effort E, small gain thr. T
Output: Optimized AIG N.

```
1: Ndup = N; / * for lossless synthesis */
2: lut_map(N, k); gain = 0;
3: while E > 0 do
4:     update(gain);
5:     if gain is flat for too many iterations then
6:         break;
7:     end if
8:     if gain > T then
9:         /* if gain > T, apply cheap optimization*/
10:        lut_reduction(N); simplify_dc(N);
11:        lut_map(N, k); reduce_budget(E);
12:        continue;
13:    end if
14:    det_randomize(N);
15:    /* apply stronger optimization 1 */
16:    lut_resub(N); lut_reduction(N);
17:    simplify_dc(N); lut_map(N, k);
18:    reduce_budget(E); update(gain);
19:    if gain > T then
20:        continue;
21:    end if
22:    /* apply stronger optimization 2 */
23:    lut_resub(N); Boolean_rewiring(N);
24:    simplify_dc(N); lut_map(N, k);
25:    reduce_budget(E); update(gain);
26: end while
27: Boolean_decomp(N);
28: if N is better than Ndup then
29:    return N;
30: end if
31: return Ndup;
```

A. LUT Engine

We propose a LUT optimization engine (called LUT-engine), capable of reducing the implementation complexity in terms of NAND2/AIG count, with efficient runtime. Algorithm 2 depicts the pseudocode. Note that some optimization parameters are omitted for the sake of brevity.

The goal is to make LUT optimization adaptive. This is achieved by using gradient computation of the NAND2/AIG count gain: It allows us to decide dynamically the best next attempted transformation. The LUT engine algorithm starts by duplicating the current network: This is done in case the LUT engine degrades the NAND2/AIG cost and needs reverting at the end. Before starting the iterative loop, a first valid k-LUT mapping of the network is obtained. The iterative loop is controlled by a budget E, which is consumed by running various LUT optimizations. The LUT optimizations are applied in a waterfall model where the first successful move is picked. This leads to better runtime as compared to parallel model, but it may overlook optimization opportunities. In the proposed LUT engine, the waterfall model is a good tradeoff between runtime and QoR. We employ three main LUT optimization moves in the waterfall model. The first move is the cheapest in terms of runtime and consists of LUT reduction, simplification with don't cares, and remapping. Until the gain exceeds the threshold T, the algorithm keeps applying this least runtime expensive move. Intuitively, this means that we can reduce the complexity of the network greatly with low effort optimization, and until this is possible, computational-intensive transforms

are not needed. If the gain is less than the threshold T, LUT moves featuring stronger optimization are attempted. Note that, before getting into the stronger moves, we apply deterministic randomization of the LUT network (line 14 of Algorithm 2). This is important to escape local minima and it includes shuffling fanin/fanout order of the nodes and picking different topological orderings. The first higher effort LUT optimization move consists of Boolean resubstitution, LUT reduction, simplify with don't cares, and LUT remapping. Here, Boolean resubstitution is the main workhorse considerably reducing the network cost. The second, and last, higher effort LUT optimization move comprises Boolean rewiring, Boolean resubstitution, simplify with don't cares, and LUT remapping. Boolean rewiring is a powerful transform to escape local minima as it can add/remove wires, reduce literal cost, and highlight new LUT compaction opportunities. It is worth mentioning that, if the gain remains flat for too long, the LUT engine would automatically bails out to save runtime. At the end of the iterative loop, we decompose the network into an AIG via algebraic and Boolean decomposition, as detailed in Section IV. Finally, if the cost of the network is better than the initial cost, we commit the change, otherise we revert.

In our experiments, we obtained the best NAND2 cost over industrial benchmarks by using E = 50 and k = 3, with T = 1% and bailout when gain $<$ 0.1% for 5 iterations or more. We observed that back to back call of LUT-engine with k = 3, 4, 5 can help QoR further at expenses of runtime: size 4 and 5 can compact more advatangeously the logic for some designs. Nevertheless, k = 3 remains the most effective LUT size for our LUT-engine. Experimentally, we also observed that using three-input nodes offers the best, and most general, tradeoff between functionality and structure for LUT-engine. An additional advantage of 3-input nodes is that this is an average size of gates in typical standard-cell libraries, thus optimizing this type of networks directly translates into improving the quality of technology mapping.

B. I2c Case Study

To showcase the effectiveness of LUT-engine standalone, we compare it to a similar methodology that only considers AIGs: AIG-engine [16]. Even though the AIG-engine methodology provides superior results with respect to static scripts, e.g., resyn2 in [25] and script.rugged in [26], it still does not catch all optimization opportunities. Fig. 1 shows the comparison of AIG-engine vs. LUT-engine for the *i2c* controller, which is a relevant benchmark of the EPFL suite [4]. The setup of AIG-engine follows the one from [16], but limited to 40 iterations. The setup of LUT-engine is k = 3, with T = 1% and fixed 40 iterations. We have found that *i2c* academic benchmark has good correlation with the trend observable in industrial designs. Considering AIG-engine, the NAND2 count for *i2c* goes from 1342 to 1065. We observe good improvements that saturate with increasing iteration number. On the other hand, LUT-engine is immedialy able to get *i2c* NAND2 count down to mid 800s, thanks to cheap moves containing lut-mapping and simplify-dc. Even after that, LUT-engine continues with steady gains and escape from local minima. The saturation eventually arrives only around

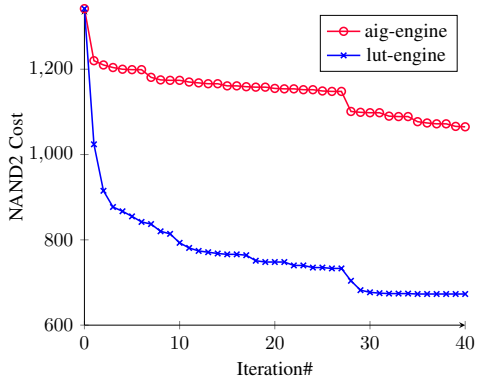

Fig. 1. Comparison of lut-engine and aig-engine on *i2c* benchmark.

iteration 30, when the flat gain would trigger the bailout mechanism, which we disabled in this experiment for the sake of comparison. The final NAND2 count for LUT-engine is 673, which translates to the smallest AIG currently known for *i2c* benchmark, improving on top of the previous best AIG result of size 710 reported in [16]. Please note that other best AIG results have been improved by LUT-engine as compared to [16]: They are not discussed here for the sake of brevity but can be downloaded at [20].

VI. Experimental Results

In this section, we present experimental results for our LUT optimization flow: LUT-engine. First, we challenge LUT-engine to improve the best results in the EPFL benchmark suite [4]. Then, we integrate LUT-engine in an industrial EDA flow, and show sensible QoR gains post place & route.

A. Methodology

We implemented our proposed methodology as part of a commercial design automation solution. In the EDA flow, LUT-engine runs after the initial Boolean optimization, which mainly aims at reducing area. Therefore, LUT-engine targets size reduction in the logic network. Tight control on the number of levels and the number of nets is enforced during LUT-engine: this is known to correlate with delay and congestion later on in the flow. To run tests on the EPFL benchmarks, we also compiled LUT-engine as a standalone package.

B. EPFL Benchmarks

In this section, we show the results for the EPFL benchmarks. In particular, we challenge the area (i.e., number of LUTs) category within the EPFL benchmark suite project that keeps track of the best 6-input LUT synthesis results. We used LUT-engine with $k = 6$ and T = 1%. Additionally, we disabled the final AIG decomposition in LUT-engine and added a final mapping using $k = 6$, without literal costing, to make sure the output network is a valid 6-LUT network. We let LUT-engine run dynamically with a maximum budget corresponding to 3 hours, and early bailout if gain $<$ 0.1% for more than 20 iterations. Note that most benchmarks run in minutes, with exception of the large ones, such as *log2* and *hypotenuse*.

We improved the previous best size (area) results[3] for the 12 benchmarks reported in Table II. Even though the EPFL

[3]The EPFL best results are available at: *https://github.com/lsils/benchmarks*. We compare our results to latest commit *7c9f16e*

TABLE II
NEW BEST AREA RESULTS FOR THE EPFL SUITE

Benchmark	I/O	6-input LUT count	Level Count.
adder	256/129	**191**	184
arbiter	256/129	**307**	78
divisor	128/128	**3250**	1189
hypotenuse	256/128	**39826**	4492
log2	32/32	**6513**	132
mem_ctrl	1204/1231	**2019**	21
mult	128/128	**4898**	93
priority	128/8	**101**	26
sin	24/25	**1205**	61
sqrt	128/64	**3030**	1093
square	64/128	**3232**	76
voter	1001/1	**1281**	19

TABLE III
POST PLACE&ROUTE RESULTS ON 36 INDUSTRIAL DESIGNS

Flow	Area	Sw. Power	WNS	TNS	Runtime
Baseline	1	1	1	1	1
LUT engine	**-1.80%**	**-1.72%**	**-0.42%**	**-0.39%**	+5%

benchmarks have been optimized several times in the last 5 years, for some of the benchmarks our improvement is larger than 500 6-LUTs. Further, we obtained a new best result for the *adder* benchmark, that was not improved since 2016. Our circuit implementations can be downloaded at [20].

C. ASIC Results

We present here our experimental results on 36 industrial ASIC benchmarks, obtained using a commercial EDA flow empowered with our new LUT-engine. Since the ASIC designs come from major electronic industries, we cannot disclose their details. To show the efficacy of our method, we thus present the average results w.r.t. a baseline flow without our LUT-engine. The results, post place & route, are summarized in Table III. All benchmarks are verified to be equivalent with an industrial formal equivalence checking flow. Our complete design flow, embedding the new LUT-engine, achieves sensible combinational area & combinational switching power reductions, 1.80% and 1.72% respectively, on average, and also WNS/TNS improvements, with moderate 5% runtime cost.

VII. CONCLUSIONS

In this paper, we developed a new LUT-based optimization flow that is tailored for the synthesis of ASICs rather than FPGAs. We enhanced LUT mapping to consider literal/AIG cost of each LUT node. We extended traditional Boolean methods to simplify and re-shape LUT-networks, targeting the best AIG/mapped-network implementation, after decomposition. Intuitively, literal-driven LUT packing behaves as an effective fanin-bounded eliminate, unveiling higher-order Boolean simplification opportunities. We embedded our proposed LUT optimization flow, area oriented, in a commercial synthesis tool. Using our methodology, we improved 12 of the best area results in the EPFL synthesis competition. Employed in a commercial EDA flow for ASICs, our LUT optimization reduced area by 1.80%, total negative slack by 0.39%, and switching power by 1.72%, after physical implementation, at 5% runtime cost.

REFERENCES

[1] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Improvements to technology mapping for LUT-based FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 240–253, 2007.

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

[3] G. Chen and J. Cong, "Simultaneous logic decomposition with technology mapping in FPGA designs," in *International symposium on Field programmable gate arrays*, pp. 48–55, 2001.

[4] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *International Workshop on Logic & Synthesis*, 2015.

[5] R. Francis, J. Rose, and Z. Vranesic, "Chortle-crf: Fast technology mapping for lookup table-based FPGAs," in *Design Automation Conference*, pp. 227–233, 1991.

[6] L. Cheng, D. Chen, and M. D. Wong, "DDBDD: Delay-driven BDD synthesis for FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1203–1213, 2008.

[7] L. Machado and J. Cortadella, "Support-reducing decomposition for FPGA mapping," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[8] P. Pan and C.-C. Lin, "A new retiming-based technology mapping algorithm for LUT-based FPGAs," in *International Symposium on Field Programmable Gate Arrays*, p. 35–42, 1998.

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

[10] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification," tech. rep., ERL Technical Report, 2005.

[11] G. Liu and Z. Zhang, "PIMap: A flexible framework for improving LUT-based technology mapping via parallelized iterative optimization," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 11, no. 4, pp. 1–23, 2019.

[12] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 264–300, 1990.

[13] K. A. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, C. R. Morrison, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Multilevel logic minimization using implicit don't cares," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 7, no. 6, pp. 723–740, 1988.

[14] S. Muroga, "Logic synthesizers, the transduction method and its extension, sylon," in *Logic Synthesis and Optimization*, pp. 59–86, Springer, 1993.

[15] L. Amarú, M. Soeken, P. Vuillod, J. Luo, A. Mishchenko, J. Olson, R. Brayton, and G. De Micheli, "Improvements to Boolean resynthesis," in *Design, Automation & Test in Europe Conference & Exhibition*, pp. 755–760, 2018.

[16] E. Testa, L. Amarú, M. Soeken, A. Mishchenko, P. Vuillod, J. Luo, C. Casares, P.-E. Gaillardon, and G. De Micheli, "Scalable Boolean methods in a modern synthesis flow," in *Design, Automation & Test in Europe Conference & Exhibition*, pp. 1643–1648, 2019.

[17] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang, "Scalable don't-care-based logic optimization and resynthesis," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 4, no. 4, pp. 1–23, 2011.

[18] G. D. Micheli, *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.

[19] O. Coudert, "On solving covering problems," in *Design Automation Conference*, pp. 197–202, 1996.

[20] Anonymous link: https://drive.google.com/drive/folders/1Vf2UdWvBBkEhTuzL2slUE7klEW4yYWZK?usp=sharing.

[21] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic minimization algorithms for VLSI synthesis*, vol. 2. Springer Science & Business Media, 1984.

[22] E. J. McCluskey, "Minimization of Boolean functions," *The Bell System Technical Journal*, vol. 35, no. 6, pp. 1417–1444, 1956.

[23] S.-C. Chang, L. P. Van Ginneken, and M. Marek-Sadowska, "Circuit optimization by rewiring," *IEEE Transactions on computers*, vol. 48, no. 9, pp. 962–970, 1999.

[24] F. Brglez, D. Bryan, J. Calhoun, G. Kedem, and R. Lisanke, "Automated synthesis for testability," *IEEE Transactions on Industrial Electronics*, vol. 36, no. 2, pp. 263–277, 1989.

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

[26] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," 1992.