Improving LUT-based Optimization for ASICs

Walter Lau Neto*, Luca Amarú†, Vinicius Possani†, Patrick Vuillod†, Jiong Luo†,
Alan Mishchenko‡, Pierre-Emmanuel Gaillardon*

*LNIS, University of Utah, Salt Lake City, Utah, USA
†Synopsys Inc., Design Group, Sunnyvale, California, USA
‡Department of EECS, University of California, Berkeley, USA

***Abstract*—LUT-based optimization techniques are finding new applications in synthesis of ASIC designs. Intuitively, packing logic into LUTs provides a better balance between functionality and structure in logic optimization. On this basis, the *LUT-engine* framework [1] was introduced to enhance the ASIC synthesis. In this paper, we present key improvements, at both algorithmic and flow levels, making a much stronger *LUT-engine*. We restructure the flow of *LUT-engine*, to benefit from a heterogeneous mixture of LUT sizes, and revisit its requirements for maximum scalability. We propose a dedicated LUT mapper for the new flow, based on *FlowMap*, natively balancing LUT-count and NAND2-count for a wide range LUT sizes. We describe a specialized Boolean factoring technique, exploiting the fanin bounds in LUT networks, resulting in a very fast LUT-based AIG minimization. By using the proposed methodology, we improve 9 of the best area results in the ongoing EPFL synthesis competition. Integrated in a complete EDA flow for ASICs, the new *LUT-engine* performs well on a set of 87 benchmarks: -4.60% area and -3.41% switching power at +5% runtime, compared to the baseline flow without LUT-based optimizations, and -3.02% area and -2.54% switching power with -1% runtime, compared to the original *LUT-engine*.**

I. Introduction

Look-up table (LUT) based techniques for logic synthesis and mapping have significantly evolved in the last decades [2]–[4], improving FPGA implementations dramatically. While these methods naturally target FPGAs, LUT-based techniques are also increasingly finding applications in logic synthesis for ASICs [1]. Packing logic into LUTs offers the opportunity to balance functionality and structure in the circuit representation. With LUTs, Boolean optimizations are more powerful, since functionality is more dense across levels, and the backbone structure of the circuit is still retained, in contrast to a more aggressive logic optimization by full or partial collapsing [5]. The *LUT-engine* framework was introduced to exploit this opportunity and enhance the previous work on ASIC synthesis [1]. It consists of iterative LUT-k mapping and LUT-k Boolean simplifications, driven by optimizing optimizing the *Number of Literals in Factored Form* (NLFF) rather than LUT counts, followed by a final AIG decomposition.

This paper continues the line of research in harnessing LUT-based techniques for ASICs. We investigate the available *LUT-engine* with the goal of developing a new *LUT-engine* capable of very fast inner-loop iterations, driven by the most scalable mappers and optimizers, but at the same time not limited by only one LUT size k for the duration of the engine. Such goal requires algorithmic and flow level innovations to build a *quality of results* (QoR) convergent framework. In particular, the contributions of this work are:

- Restructuring the *LUT-engine* flow to accommodate heterogeneous values of k. Revisiting the sequence of optimizations/mapping/network-transformations to maximize QoR and guarantee convergence, especially in the presence of mixed-k LUTs. We also propose methods to pick heterogeneous LUT sizes for best QoR.
- Introducing a new LUT mapper based on *FlowMap* [6] that is more scalable than the previous one based on binate covering. The new mapper natively balances the LUT count and the NAND2 count. This allows for larger sizes of k to be used while maintaining a tight control on the NAND2 impact, which is key for QoR improvement.
- Developing a specialized Boolean factoring, based on the known bounds on the nodes' fanin in LUT networks. This allows for a quick and scalable decomposition of LUTs into minimized AIGs, a crucial step to achieve fast inner-loop iterations in the new *LUT-engine*.

Altogether the new *LUT-engine* runs its inner loop up to one order of magnitude faster than the baseline. Moreover, it reaches deeper into the solution space, leading to the improved QoR. The proposed new *LUT-engine* is tested on both academic and industrial benchmarks. When targeting the ongoing EPFL synthesis competition, we improved 9 of the best known area results. We embed our new *LUT-engine* in a commercial EDA tool and test it on 87 RTL benchmarks: 22 public OpenCore designs and 65 industrial designs. On average, our new flow enables -4.60% area and -3.41% switching power at +5% runtime, compared to the baseline flow not employing LUT-based optimizations. When compared directly with the original *LUT-engine* [1], the new flow shows -3.02% area and -2.54% switching power with -1% runtime.

The paper is organized as follows: Section II presents relevant background. Section III proposes a high-level flow of LUT-based optimization. Section IV introduces a new scalable LUT mapper and explains how it is modified for the use in the proposed *LUT-engine*. Section V details the Boolean factoring procedure employed. Section VI shows experimental results for EPFL synthesis competition and comparison with a commercial EDA tool. Section VII concludes this work.

II. Background

A. Circuit Representation

A logic circuit can be naturally represented as a directed acyclic graph (DAG), denoted by $G = (N, E)$, where each node $n \in N$ may have incoming (*fanin*) edges. Nodes without fanins are *primary inputs* (PIs) while nodes with fanins implement Boolean functions. An outgoing edge of a node n is the node's *fanout*. Edges may or may not be complemented [7]. A common type of DAG for logic manipulation is an *and-inverter graph* (AIG) where each node has the Boolean function of a two-input AND.

B. The Original LUT-Engine

LUT-engine maintains a synergistic balance between functionality and structure while running LUT-based logic synthesis methods for ASIC designs. The original LUT-engine framework performs three main steps: (i) mapping an AIG into a LUT network, (ii) running LUT-based optimizations, and (iii) decomposing the LUT network back into an AIG.

The first step derives a valid LUT mapping of the initial circuit by performing a pass of tree mapping [8] followed by DAG mapping. The DAG mapping performs *k*-cut enumeration and builds a binate covering problem [9] solved by a branch-and-bound approach. The value of *k* defining the maximum LUT size is given by the user. The same value of *k* is used throughout the LUT-engine flow for both LUT mapping and LUT-based optimizations. The mapping uses the NLFF as a cost function when targeting ASIC implementations.

Then, in a waterfall way, the algorithm dynamically chooses among three efforts of optimizations to run over the mapped LUT network: low effort, medium effort, and high effort. While the network size is being improved from one iteration to another, the algorithm applies low-effort optimizations. If the gain is flat for a defined number of iterations, it uses more aggressive optimizations as needed. The authors rely on a gradient-based computation of the AIG size to set the gains and choose between the three levels of optimization effort.

Optimizations done at the LUT level aim to reduce the LUTs complexity, and use a specialized cost function to optimize the NAND-2 implementation. This is because, at the end of the loop, the LUT-network is decomposed into an AIG, to be mapped into standard cells. Before mapping, the AIG is optimized using the traditional AIG optimization passes, such as rewriting, resubstitution, and refactoring [10]. The optimized AIG is then tech-mapped for ASICs, resulting in substantial QoR improvements.

III. Restructuring the LUT-engine

We propose a new flow for LUT-based optimization targeting ASIC designs, with two aspects in mind: (i) achieving better *Power, Performance, and Area* (PPA) and (ii) having better scalability. To this end, we present and discuss the key improvements and differences of the new LUT-engine.

The first key difference is that this work adopts different LUT granularities during the optimization flow. Let *k* be the number of LUT inputs. Whereas the previous LUT-engine assumes a single *k* for the whole flow, we allow for a more diverse set of LUT sizes. In practice, our flow considers both smaller and bigger sizes of LUTs, compared to the previous one. That may allow the proposed flow to strike a better balance between functionality and structure on the fly. Thus, every iteration uses a different *k* value. In this context, larger *k* values provide powerful Boolean optimization opportunities to reduce the complexity of the functions, whereas smaller sizes unlock an iterative restructuring of the network. The values of *k* may be chosen in different ways.

Furthermore, at every iteration, after doing optimization at the LUT level, the mapped LUT network is decomposed into an AIG, and AIG-based optimization takes place. This is in contrast to the previous LUT-engine that decomposes the LUT network into an AIG only at the end of the optimization loop. Therefore, previously the functionality of the circuit was optimized iteratively, and the structural optimization was done using expensive AIG decomposition.

As the proposed flow decomposes the LUT network into an AIG and optimizes it in every iteration, we need both: (i) powerful and efficient AIG decomposition and optimization methods; and (ii) a specialized LUT mapper that optimizes an AIG-oriented cost function for different *k* sizes, resulting in a quick runtime. This is important to ensure that the new flow still converges towards a good AIG implementation, even for a large range of different LUT sizes. This is a different goal compared to reducing the number of LUTs in each iteration. Indeed, we might increase the number of LUTs from one iteration to another and yet have a better AIG implementation cost.

Algorithm 1 presents a high-level view of the new LUT-engine flow. The inputs are an AIG network $\mathcal{N}$, an effort level $\mathcal{E}$, which in practice defines how many iterations are performed, and a strategy $\mathcal{S}$ to chose the values of k. First, the LUT-engine duplicates the input AIG and stores it in the $\mathcal{N}'$ variable, which holds the best seen AIG. Then, an auxiliary function sets the value of k for the LUT mapping and related optimizations. Different strategies can be employed to select *k*, as shown in Algorithm 2. Depending on the user input, the engine can increment *k* by one, apply pseudo-randomization based on a random seed provided by the user, iterate over a pre-defined list of values, or increment *k* by different amounts, not only one. Our flow adopts a custom function that increases *k* in different ways and selects it using a pseudo-random strategy $\mathcal{S}$.

Algorithm 1: New LUT-engine Flow

Input: AIG $\mathcal{N}$, Effort level $\mathcal{E}$, Strategy $\mathcal{S}$, Maximum *k* size *m*, Seed *x*, List of sizes *l*
Output: Optimized AIG $\mathcal{N}'$

```
N' ← duplicate(N);
while E > 0 do
    k ← select_k(S, m, k, x, l);
    compute_choices(N);
    BoundedNtk ← new_specialized_mapper(N, k);
    boolean_optimization(BoundedNtk);
    N ← boolean_decomposition(BoundedNtk);
    aig_optimization(N);
    if num_nodes(N') > num_nodes(N) then
        N' ← duplicate(N);
    update_effort(E);
return N';
```

Next, we compute structural choices used to reduce structural bias during technology mapping [2]. The input for the choice computation is a decomposed and optimized AIG network from several previous flow iterations. The main idea of computing choices nodes is to group nodes with the same functionality into equivalent classes given to the technology mapper. A SAT-based approach is adopted to compute choices nodes that work on an AIG structure [2]. Thus, it is crucial to

Algorithm 2: k Size Selection Algorithm

Input: Strategy $\mathcal{S}$, Maximum k size m, Seed x, List of sizes l
Output: LUT size k

```
k ← 0;
if S == 0 then
    k ← increment_by_one(k, m);
else if S == 1 then
    k ← deterministic_randomization(k, m, s);
else if S == 2 then
    k ← iterate_over_list(k, l);
else
    k ← customized_value(k, m);
return k;
```

have an AIG as small as possible to reduce the overall runtime, justifying our need for an efficient Boolean decomposition.

Once the choice nodes are computed, the flow runs a specialized LUT mapping tailored to ASICs. Since we run technology mapping at every iteration, it is key to have an efficient mapper that delivers good quality mapping in short runtime. Thus, we propose a cut-based structural mapper detailed in Section IV. This is different from the previous LUT-engine, which relies on binate covering. After technology mapping, the network goes through a sequence of Boolean transformations, which optimize the *functionality* of the network. Several methods are used. We quickly review these in Section III-A.

The next step is to decompose the LUT network into an AIG. For this, we propose to use a Boolean decomposition tuned to consider the AIG implementation cost. That is another key difference compared to the new LUT-engine because it does not use algebraic factorization to decompose the network. As the proposed flow works with LUTs up to 16 inputs, truth-table-based methods can be employed with affordable runtime. Details about the Boolean decomposition algorithm are presented in Section V-A.

After decomposing the LUT network into an AIG, a series of AIG optimizations is performed. These optimizations include four passes: (i) rewriting, (ii) refactoring, (iii) resubstitution, and (iv) balancing. The idea is to optimize the circuit structure, reducing the circuit size and depth. Finally, the algorithm checks to see if the final network is better than the initial network using the NAND2 gate count as a cost function. If so, it accepts the transformations and updates the number of iterations.

A. Boolean Optimization for LUT-Networks

This section quickly presents and reviews the methods we adopt in our LUT-Optimization framework targeting ASICs. There are three main steps: (i) LUT complexity minimization, (ii) Boolean resubstitution, and (iii) Boolean re-wiring for LUTs.

1) LUT Complexity Minimization: LUT complexity reduction aims to reduce the ASIC implementation cost, post-AIG decomposition, and standard cell mapping. Our proposed flow relies upon two main methods to reduce the intrinsic LUT complexity: (i) 2-level *Sum of Products* (SOP) minimization exploiting don't cares and (ii) support reduction. The first enables powerful don't care based optimization thanks to a tight boundary on the fanin given by the LUTs. That allows us to exploit *controllability don't care* (CDC) and *observability don't care* (ODC) efficiently without dramatically increasing the flow runtime. We use as cost function to evaluate the SOP minimization the factored literal cost, as it has a better co-relation with the ASIC implementation. The support reduction goal is to remove redundant inputs that LUTs might have when considering the global network functionality. In this sense, each LUT support is computed using either BDD or SAT methods [3], [11], [12]. We can delimit a frontier to keep the methods scalable. If the support size is smaller than the current LUT, the LUT SOP is replaced by a new ISOP.

2) Boolean Resubstitution: Boolean resubstitution can work remarkably well in LUT networks, as it allows for more savings and high-order optimization opportunities, compared to AIG networks [3], [13]. Here we start by sorting the LUTs by their potential gains. The considered gain is the NLFF that this LUT saves if removed, computing while taking into account the *Maximum Fanout-Free Cone* (MFFC) of each LUT. We then iterate over the LUTs, computing their MSPF (flexibilities) and trying to re-express the LUT functionality by leveraging other LUTs already presented in the network. This set of candidate nodes to re-express the original LUT functionality can be computed using BDDs, SAT, or truth-tables [3]. Our method relies mostly upon a SAT-based approach to compute the resubstitution candidates.

3) Boolean Re-wiring: Boolean re-wiring iteratively adds and removes redundant wires in the network. The idea is that adding a new redundant wire might lead to more wires being redundant under certain conditions, then leading to new optimization opportunities [14]. This is a multi-node optimization, differing from Boolean resubstitution and LUT simplification, which considers one node at a time. The rewiring can be done by adding new gates or by replacing existing ones using gates with more inputs. In the sequence, ATPG is applied to identify possible redundancies in the modified network [15]. The final goal of our Boolean re-wiring is to reduce the NLFF of the network implementation.

IV. Application-Specific Mapper for LUT-Engine

The main goal of the new application-specific LUT mapper tailored to ASICs is to be highly scalable while delivering a good quality of results. In particular, we want the mapper to converge to a good NAND-2 gate count while working with different values of k. That allows the proposed LUT-engine to converge to a small AIG implementation when using a wide range of LUT sizes.

The adopted mapper has six major steps: (i) computing structural choices, (ii) computing k-cuts, (iii) computing the cuts truth-table and NAND-2 equivalent cost, (iv) setting the best cut for each node based on the selected cost function, (vi) computing the network cover, and (vi) performing area recovery for the mapped network. Below, we focus on explaining the cut computation and efficient cut storage used. We then present our heuristic to select the best cut. We also discuss

how we adapt area flow for area recovering. For a more in-depth review on choice computation and network covering, we refer the reader to [7], [16].

The first step is to compute the *k*-cuts. A cut c of a root node n is a pair (n, L), where L is the set of cut leaves, such that any path from a PI to n passes through at least one leaf. A *trivial* cut of n is composed of n itself. Non-trivial cuts cover all the nodes found on the paths from the root to the leaves, including the root and excluding the leaves. A cut is *k*-feasible if it has up to *k* leaves. Thus, starting from the PIs, and given an internal node n with fanins $a, b \in N$, the set of cuts for n, $\Phi(n)$, can be obtained through the *Union* of the cut sets from a and b as follows:

$$\Phi(n) = \{\{n\}\} \cup \{u \cup v | u \in \Phi(a), v \in \Phi(b), |u \cup v| \leq k\} \quad (1)$$

During the *k*-cut enumeration, the Boolean function and the area cost of each cut are computed on the fly. The functions are compactly stored as truth tables, since the network has a limit in the number of LUT inputs given by *k*. The area cost is given by the literal count in the factored form [17]. A factorization method is used for area estimation. A good factorization computation is key to enable good QoR. Still, as it is expansive in runtime, we compute it only a single time for every cut and cache this information to improve the mapper runtime. For every cut, we store: (i) the unique IDs of the cut leaves, (ii) the cut truth table, and (iii) the cut area cost computed using its factored form. All information is represented using integer numbers. Thus, for every node, its cut set can be quickly retrieved using array indexing where the index is given by the node ID. Similarly, as the number of cuts in the cut set is known, it is possible to access any cut of any node using array indexing. Therefore, reading the cut information is done in constant ($\mathcal{O}(1)$) time.

To define the best cut during the network covering, we use an adaptation of the area flow heuristic (AF) [18], [19]. The area flow provides an estimate of the area cost for a LUT taking into account the logic sharing of the LUT inputs and outputs. In the usual case for a generic LUT mapper, the area used for every LUT during area flow is considered to be unitary, i.e., all the *k*-input LUTs have the same area. In our case, the area is given by the cached factored literal count. Based on the best cut, the covering is performed. The caching helps in saving runtime as computing the factored form of the cuts has runtime costs, and the new mapper achieves great scalability compared to the previous binate covering approach. Relying on a good factorization for area estimation, we achieve strong gains in PPA as discussed in Section VI.

Once the network is covered, the proposed flow performs several iterations of area recovery based on our adaptations of area flow and the exact (local) area [2]. The exact area of a cut rooted at a node n is usually computed by summing up the number of LUTs needed to implement the MFFC of n, since all *k*-input LUTs are assumed to have the same area cost. In our case, we compute it by summing up the factored literal cost of the LUTs in the MFFC of n. The exact area gives a local view of how many literals could be saved by removing the LUT (cut) in the mapping. Our flow iterates the area recovery using exact local area and area flow. The number of iterations of each area recovery strategy can be controlled to trade off QoR and runtime.

Altogether, these key enhancements enable our flow to run LUT mapping iteratively while keeping the runtime under control and converging to a better NAND2-equivalent implementation for different LUT sizes. A runtime profile of our new mapper shows it takes a very small fraction of the flow runtime, being an order of magnitude faster than the mapper used in the original LUT-engine [1], which is helpful for speeding up LUT-based optimizations.

V. Scalable Boolean Decomposition for LUT-engine

This section presents and discusses a scalable Boolean decomposition tailored to ASICs. Boolean factorization is a crucial step of the proposed LUT-engine, since it enables efficient iterative AIG decomposition while reducing the ASIC implementation cost.

The logic division is a critical component of many techniques [7]. In general terms, given two Boolean functions, f and p, the division of f by p gives a quotient q and a remainder r. Thus, $f = p.q + r$, where the sign . denotes the AND Boolean operator, and the sign $+$ denotes the OR Boolean operator. If the remainder is not null, then p is a Boolean divisor of f, otherwise, p is said to be a Boolean factor of f. Note that, in the general case, q and r are not unique, and a function might have several Boolean divisors and factors, which introduces a problem of selecting a good divisor. A restricted version of division is the algebraic division, where the quotient is unique. It restricts the function to be minimal with respect to single cube containment, i.e., there is not a single cube containing another cube in the expression. An algebraic product $p.q$ is such that the supports of p and q are disjoint, i.e., p and q have no variables in common. These concepts are used in our flow to explain the Boolean division, and how it fits in the Boolean decomposition algorithm. We use the terms *weak* and *algebraic* division interchangeably.

A. Boolean Decomposition

Boolean decomposition can be done in a number of ways. For instance, Boolean functions of the circuit outputs can be derived and decomposition can be applied to these functions [20]. However, this has a high complexity, and collapsing is not always feasible. Another possibility is to run Boolean decomposition to reduce the support of LUTs [12]. In our case, reducing the LUT support is handled by our engine during the Boolean optimization of LUT networks (see Section III-A). Other methods rely on disjoint support decomposition [21], and are less general because this decomposition does not always exist. Therefore, in this work, we use a new specialized Boolean decomposition that aims at reducing the factored literal cost. This approach has a better correlation with our goal to minimize the post decomposition NAND2-equivalent implementation cost, instead of the LUT count.

The proposed algorithm based on Boolean division is described in Algorithm 3. Boolean division is more powerful than algebraic division and it can be achieved by tweaking the weak division. In the proposed approach, a new variable can be added to replace p (the divisor), i.e., let's say x,

and the expression $p \oplus x$ can be introduced as a don't care condition for f. The $\oplus$ refers to the Exclusive-OR Boolean operator. The function f is then minimized with a two-level logic minimizer that considers the *satisfiability don't care* (SDC) condition introduced. Finally, weak division is run in the resulting optimized function. In our proposed Boolean decomposition flow, we leverage Boolean division.

Initially, the decomposition algorithm iterates over the LUTs and retrieves their Boolean functions expressed as truth tables. Since we are considering up to 16 input LUTs, truth table computation can be handled efficiently with integer-based implementation. For larger sizes, SAT engines can be used. To have a proxy of each LUT complexity, we consider the NLFF as the LUT cost.

In the sequence, the algorithm compute the candidate divisors based on kernels to decompose the LUT. Kernel intersection is performed to find candidate divisors that improve sharing among LUTs [16]. As the input LUT network has a tight boundary in the fanin number, kerneling computation and intersection can be run with acceptable runtime. Nevertheless, our framework lets us control the level of kernels and the number of divisors to try.

Then, for each potential divisor, Boolean division takes place. A new variable x is introduced in the function, and the Exclusive-OR of this variable with the divisor is projected as the SDC in the don't care conditions of the function along with the function's don't cares. The gain of the division is given by how many literals are saved compared to the original implementation of f. The best divisor is the one having the biggest reduction in the literal count.

Once the best divisor is selected, each LUT is decomposed to build a new AIG. Alternatively, if the division does not improve the AIG implementation, the algorithm decomposes f itself. Structural hashing takes place to increase logic sharing whenever possible while creating new AIG nodes.

Algorithm 3: Boolean Decomposition Flow

Input: LUT-Mapped network $\mathcal{N}$
Output: Decomposed AIG $\mathcal{N}'$

```
for each LUT L in topological order do
    f ← lut_fucntion(L);
    current_cost ← factored_form(f);
    D ← compute_recursively_divisors(f);
    best_divisor ← ∅;
    for each d ∈ D do
        DC ← DC(f) + (x ⊕ d);
        f' ← two_lvl_minimizer(f, DC);
        new_cost ← factored_form(f');
        if current_cost > new_cost then
            current_cost ← new_cost;
            best_divisor ← d;
    if d != ∅ then
        build_aig(N', d);
        continue;
    build_aig(N', f);
return N';
```

VI. Experimental Results

This section presents experimental results for our new *LUT-engine*, starting with the new best results in the ongoing EPFL synthesis competition [22]. Next, we integrate our *LUT-engine* in a commercial EDA flow. We show substantial improvements in QoR, post place & route, and better runtime and scalability, using the original *LUT-engine* [1] as the baseline.

A. Methodology

We integrate the proposed *LUT-engine* in a commercial design flow. The new *LUT-engine* runs during Boolean optimization and logic synthesis, driven by area minimization goal. In addition to measuring area improvements, we also control the level count and the net count. These metrics are known to correlate with delay and congestion in the physical synthesis portion of the design flow. To consider EPFL benchmarks, we compiled the proposed engine as a standalone package capable of processing LUT networks.

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 obtained by the participants since the beginning of the competition in 2015 [22]. We use the proposed new *LUT-engine* with k up to 10, random $\mathcal{S}$ strategy, and runtime budget of 60 minutes, with early bailout when no gain/change in the network cost is observed for 5 consecutive loops. As final step, we add a $k = 6$ mapping with no ASIC considerations, targeting minimum numbers of LUT-6, as mandated by the EPFL competition. Most benchmarks run in minutes, with exception of the larger ones, such as *div* and *hypotenuse*.

TABLE I
New Best Area Circuits For The EPFL Synthesis Competition

Benchmark	I/O	6-input LUT count	Level Count.
sin	24/25	**1198**	60
hypotenuse	256/128	**39593**	4438
i2c	147/142	**198**	12
cavlc	10/11	**67**	3
arbiter	256/129	**305**	78
sqrt	128/64	**3026**	1087
voter	1001/1	**1280**	19
divisor	128/128	**3238**	1185
mem_ctrl	1204/1231	**1986**	22

We improve the previous best size (area) results[1] for 9 benchmarks reported in Table I. Our improvements range from few hundreds LUT-6 to few tens of LUT-6 or just a few LUT-6 for smaller benchmarks. Interestingly, for the first time, circuits *i2c* and *mem_ctrl* reach sizes < 200 and < 2000 LUT-6, respectively. This is significant because these circuits are known to have good correlation with industrial designs. Our circuit implementations can be downloaded at [23].

C. ASIC Synthesis Results

We present experimental results for 87 ASIC benchmarks, including 22 OpenCore designs [24] and 65 industrial designs. The OpenCore designs are *ac97_ctrl, pci_bridge32,*

[1]The EPFL benchmark and best results are available at: *https://github.com/lsils/benchmarks*. We compare our results to the latest commit *d783ca5*

aes_core, pci_conf_cyc_addr_dec, des_area, pci_spoci_ctrl, des_perf, sasc, ethernet, simple_spi, mem_ctrl, spi, ss_pcm, steppermotordrive, systemcaes, systemcdes, tv80, usb_funct, usb_phy, vga_lcd, wb_conmax, and *wb_dma*. Since the 65 industrial designs come from major semiconductor companies, with proprietary standard cell libraries and constraints, we cannot disclose their details. To show the improvements of the proposed flow, our implementation is compared against the results obtained by the original flow [1]. We have setup the original and the new *LUT-engines* to have similar runtime budgets, modulo difference in early bailouts. This way, the comparison is focusing on QoR, assuming the same optimization effort. Both original and new *LUT-engines* are compared with the baseline flow considered state-of-the-art commercial EDA flow without a *LUT-engine*. The results for OpenCore designs, post place & route, are summarized in Table II.

TABLE II
POST PLACE&ROUTE RESULTS ON 22 OPEN CORES DESIGNS

Flow	Area	Power	WNS	TNS	Runtime
Baseline	1	1	1	1	1
Original flow	-1.42%	-1.29%	+0.19%	+0.74%	+5%
New flow	**-3.23%**	**-3.73%**	**-3.41%**	**-7.15%**	+4%

We can see that the new *LUT-engine* achieves more area reduction while taking less time. When using original the *LUT-engine* as the baseline, the proposed flow provides -1.84% area and -2.47% switching power, better timing, with about -1% runtime. The results for industrial designs, post place & route, are summarized in Table III. The table shows that

TABLE III
POST PLACE&ROUTE RESULTS FOR 65 INDUSTRIAL DESIGNS

Flow	Area	Power	WNS	TNS	Runtime
Baseline	1	1	1	1	1
Original flow	-1.70%	-1.31%	-0.36%	+1.01%	+6%
New flow	**-5.06%**	**-5.41%**	**-3.09%**	**-3.16%**	+5%

the new *LUT-engine* achieves much smaller area in less time. When using original *LUT-engine* as the baseline, the proposed flow provides -3.42% area and -2.57% switching power, better timing, and -1% runtime. On a set of OpenCore and industrial designs, the new *LUT-engine* produces -4.60% area and -3.41% switching power with +5% runtime, compared to the baseline flow, and -3.02% area and -2.54% switching power with -1% runtime, compared to the original *LUT-engine*. All results are verified using an industrial formal equivalence checking flow.

VII. CONCLUSIONS

This paper continues the line of research to enable the use of LUT-based techniques in logic synthesis for ASIC designs. We presented key improvements to the recent *LUT-engine* [1], resulting in a new *LUT-engine*. The new engine exploits heterogeneous LUT sizes and leads to a 10× scalability improvement. We introduced innovative LUT mapper and Boolean factoring techniques, tailored to work as part of the new flow. Our new *LUT-engine* improved 9 of the best area results in the ongoing EPFL synthesis competition. Integrated in an EDA flow for ASICs, our new *LUT-engine* resulted in substantial QoR improvements on a set of 87 benchmarks: -4.60% area and -3.41% switching power with +5% runtime, compared to a baseline flow not using *LUT-engine* technology, and -3.02% area and -2.54% switching power with -1% runtime, compared to the original *LUT-engine*.

ACKNOWLEDGMENT

This research is partially sponsored by DARPA, under agreement number FA8650-18-2-7849.

REFERENCES

[1] L. Amaru, V. Possani, E. Testa, F. Marranghello, C. Casares, J. Luo, P. Vuillod, A. Mishchenko, and G. D. Micheli, "LUT-based optimization for ASIC design flow," in *2021 DAC*, 2021.
[2] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Improvements to technology mapping for LUT-based FPGAs," *IEEE TCAD*, vol. 26, no. 2, pp. 240–253, 2007.
[3] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang, "Scalable don't-care-based logic optimization and resynthesis," *ACM TRETS*, vol. 4, no. 4, pp. 1–23, 2011.
[4] 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.
[5] J. Cortadella, "Timing-driven logic bi-decomposition," *IEEE TCAD*, vol. 22, no. 6, pp. 675–685, 2003.
[6] J. Cong and Y. Ding, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE TCAD*, vol. 13, no. 1, pp. 1–12, 1994.
[7] G. D. Micheli, *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.
[8] R. Francis, J. Rose, and Z. Vranesic, "Chortle-crf: Fast technology mapping for lookup table-based FPGAs," in *DAC*, pp. 227–233, 1991.
[9] O. Coudert, "On solving covering problems," in *DAC*, pp. 197–202, 1996.
[10] H. Riener, E. Testa, W. Haaswijk, A. Mishchenko, L. Amarù, G. D. Micheli, and M. Soeken, "Scalable generic logic synthesis: One approach to rule them all," in *2019 56th ACM/IEEE DAC*, pp. 1–6, 2019.
[11] 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 *DATE*, pp. 1643–1648, 2019.
[12] L. Machado and J. Cortadella, "Support-reducing decomposition for FPGA mapping," *IEEE TCAD*, 2018.
[13] L. Amarú, M. Soeken, P. Vuillod, J. Luo, A. Mishchenko, J. Olson, R. Brayton, and G. De Micheli, "Improvements to Boolean resynthesis," in *DATE*, pp. 755–760, 2018.
[14] 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.
[15] 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.
[16] 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.
[17] R. Brayton and C. McMullen, "The decomposition and factorization of boolean expressions," in *International Symposium on Circuits and Systems*, pp. 29–54, 1982.
[18] V. Manohararajah, S. Brown, and Z. Vranesic, "Heuristics for area minimization in lut-based fpga technology mapping," *IEEE TCAD*, vol. 25, no. 11, pp. 2331–2340, 2006.
[19] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient fpga mapping solution," in *Proc. FPGA '99*, pp. 29–35, ACM Press, 1999.
[20] N. Modi and J. Cortadella, "Boolean decomposition using two-literal divisors," in *Proceedings of the 17th VLSID*, (USA), p. 765, IEEE Computer Society, 2004.
[21] A. Mishchenko, R. Brayton, and S. Chatterjee, "Boolean factoring and decomposition of logic networks," in *2008 IEEE/ACM ICCAD*, pp. 38–44, 2008.
[22] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *International Workshop on Logic & Synthesis*, 2015.
[23] Link: https://tinyurl.com/dac22-results.
[24] OpenCores: https://opencores.org.