Improvements to Boolean Resynthesis

Luca Amarú*, Mathias Soeken†, Patrick Vuillod*, Jiong Luo*,
Alan Mishchenko‡, Janet Olson*, Robert Brayton‡, Giovanni De Micheli†

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

***Abstract*—In electronic design automation Boolean resynthesis techniques are increasingly used to improve the quality of results where algebraic methods hit local minima. Boolean methods rely on complete functional properties of a logic circuit, preferably including don't care information. Computationally expensive engines such as truth tables, SAT and binary decision diagrams are required to gather such properties. The choice of the engine determines the scalability of Boolean resynthesis. In this paper, we present improvements to Boolean resynthesis, enabling more optimization opportunities to be found at the same or smaller runtime cost as compared to state-of-the-art methods. Our contributions include (i) a theory of Boolean filtering to drastically reduce the number of gates processed and still retain all possible optimization opportunities, (ii) a weaker notion of maximum set of permissible functions, which can be computed efficiently via truth tables, (iii) a generalized refactoring engine that supports multiple representation forms, and (iv) a practical Boolean resynthesis flow, which combines the techniques proposed so far. Using our Boolean resynthesis on the EPFL benchmarks, we improve 10 of the best known area results in the synthesis competition. Embedded in a commercial EDA flow for ASICs, the Boolean resynthesis flow reduces the area by -2.67% and total negative slack by -5.48%, after physical implementation, at negligible runtime cost.**

I. Introduction

Boolean resynthesis aims at improving an existing logic network implementation. The methods used in Boolean resynthesis are capable of stronger optimization than algebraic techniques, which usually generate the initial logic network implementation [1], [2]. Since Boolean methods have higher computational complexity than algebraic methods, they are used cautiously in *electronic design automation* (EDA) flows [1], [2].

This paper presents improvements to Boolean resynthesis. We revisit fundamental data structures and algorithms for Boolean resynthesis, in light of modern computing capabilities, enabling more optimization opportunities to be found at the same or smaller runtime than state-of-the-art methods [3], [4]. The main contributions of this paper are:

1) a theory of Boolean filtering, to *drastically reduce* the number of gates processed by resubstitution, while still retaining all possible optimization opportunities;
2) a weaker notion of the maximum set of permissible functions, called *forward functional flexibility*, which can be computed easily via truth tables;
3) a generalized refactoring engine which supports multiple representation forms and maximizes nodes sharing; and
4) a practical Boolean resynthesis flow, which combines the techniques proposed so far into a global optimization engine, for use in commercial EDA tools.

We evaluate our techniques both on academic and industrial benchmarks. By applying Boolean resynthesis directly on the LUT-6 mapped networks, we improve 10 of the best known area results in the EPFL synthesis competition [5]. We embed the Boolean resynthesis engine in a commercial EDA tool. After physical implementation, our enhanced EDA flow reduces the area by -2.67%, leakage by -2.11%, worst negative slack by -0.44%, and total negative slack by -5.48%, at negligible runtime cost.

II. Background and Motivation

A. Boolean Logic Optimization

Logic optimization methods are usually divided into two groups: Algebraic methods, which are fast, and Boolean methods, which are slower but achieve better results [1], [2]. Traditional algebraic methods treat a logic function as a polynomial. Instead, Boolean methods handle the true nature of a logic function using Boolean identities as well as don't cares to get a better solution.

While many different Boolean methods exist, this paper focuses on *resubstitution*, *rewriting*, *refactoring*, and *permissible functions*, because they show the best quality of results to runtime tradeoffs in an industrial synthesis environment. We refer the interested reader to [1], [2], [6] for a more exhaustive review on the literature for Boolean methods. We refer the reader to [2] for standard notions in logic optimization and synthesis.

1) Resubstitution: Resubstitution, or *resub* in short, (re)expresses the function of a node n using other nodes already present in the network. A resub transformation is accepted if the new implementation is better, w.r.t. a target metric, than the current node implementation using the immediate fanins. This approach generalizes to k-resubstitution, which adds exactly k new nodes and removes l nodes, where l is the number of nodes in the *maximum fanout free cone* (MFFC, [7], [8]) of n. If $l > k$, global size improvement is achieved. The functionality of the new nodes can be drawn from a library of available primitive gates for resubstitution. This approach is similar to technology-dependent resynthesis based on resubstitution [9], [10]. In its AIG implementation [7], [8], resubstitution only adds two-input AND nodes, with possibly complemented inputs/outputs.

2) Rewriting: Rewriting is a greedy algorithm which minimizes the size (and/or depth) of a logic network by iteratively selecting network subgraphs rooted at a node and replacing them with smaller precomputed subgraphs, while preserving the functionality at the root node. Typical precomputed subgraphs cover all 4 variables functions, or if applicable their more compact 222 NPN classes [7], [11], [12].

3) Refactoring: Refactoring can be seen as a variant of rewriting. It iteratively selects large cones of logic rooted at a node. Then, it tries to replace the logic structure of the cone by

978-3-9819263-0-9/DATE18/©2018 EDAA

a factored form of the root function. The change is accepted if there is an improvement in the selected cost metric (usually number of gates in the network) [7], [8].

4) Permissible Functions: Due to observability and controllability don't cares in a logic network, often the function at a node n, say f_1, can be changed to another function, say f_2, without affecting the intended behavior at the primary outputs [13], [14]. Such function f_2 is called *a permissible function* for node n. We call the set of *all* permissible functions for a node n its *maximum set of permissible functions* (MSPF). A known subset of MSPF is called *compatible set of permissible functions* (CSPF), which is characterized by the following property: if a node's function is replaced by one of its CSPF, then all other node's CSPFs remain valid, i.e., do not need recomputation.

B. Motivation

Boolean optimization methods are renown to be powerful but runtime expensive. This limits their wide applicability in automated design, thus potentially leaving optimization opportunities unseen. This work aims at spotting more Boolean optimization opportunities, without spending more runtime, thanks to a revisitation of fundamental data structures and algorithms in Boolean resynthesis.

III. Fast Exploration of the Boolean Space

This section introduces approaches for the fast exploration of the Boolean optimization search space. First, it presents Boolean filtering that aims at strongly reducing the number of candidates for resubstitution, without losing any optimization opportunity. Then, it discusses a weaker notion of MSPF, that can be easily obtained via truth tables, and details its main computation steps.

A. Boolean Filtering

In principle, k-resubstitution can produce optimal logic networks, when k is large enough. Unfortunately, k-resubstitution is computationally expensive; a limitation that already becomes apparent for $k = 1$. In order to illustrate this problem, let us first consider a typical execution scenario.

Since resubstitution is typically applied to small and medium size windows of logic, one can quickly compute each node's truth table (up to 15 inputs) or BDD [15] (up to 20 inputs). These function representations efficiently support functional equivalence checking, which are the primitive operations in a resubstitution algorithm. The goal is to keep the amount of equivalence checks low. Consider as an example 1-resubstitution using the AND-2 gate library, i.e., if we have N internal nodes in the window (with a non-trivial MFFC), we try to express each one of these nodes as the AND of two other nodes in the window. Therefore, in the worst case $O(N^2)$ equivalence checks are needed for each node, summing up to $O(N^3)$ checks for the whole window. Structural filtering can be applied, e.g., skip candidates in the transitive fan-out code of the current node, skip disjoint trees, or skip nodes whose level is too far away. Even with all structural filters, the perceived runtime complexity remains very high, which requires to set a maximum number m of candidates to be tried, at least for the first new input. In this way, the overall number of equivalence checks decreases to $O(m \cdot N^2)$. Experimentally, it is known that by setting a maximum number of candidates *we overlook many advantageous optimization opportunities*, but this becomes even more necessary when attempting resubstitution with higher order k or richer gate libraries. Despite the speedups above, many equivalence checks are still spent in verifying *null candidates*, which are nodes that cannot possibly lead to a valid solution.

In this work, we address the filtering problem for *resubstitution* from a different perspective. We make use of *canalizing functions* [16], [17] to implement Boolean filtering rules, which can exclude a whole set of candidate pairs by just inspecting a single operand. We say that a function g is p/q-*canalizing* in x if the co-factor $g_{x=p} = q$ for two constant Boolean values p and q. In other words, g's value is uniquely determined by assigning x to either 0 or 1.

Theorem 1: Let f be the function of a node we want to resubstitute using a k-input gate with function $g(x_1, x_2, \ldots, x_k)$ and k candidate nodes that have functions $n_1, n_2, \ldots, n_k$, respectively. If g is p/q-canalizing in x_i, then n_i can only be a valid candidate, if $(n_i = p) \rightarrow (f = q)$, or logically equivalent $(f \neq q) \rightarrow (n_i \neq p)$.

Example 1: The two input AND function is $0/0$-canalizing in both its inputs. Assume we want to express some node with function f using two other nodes with functions a and b. Then a can only be valid, if $f \rightarrow a$, or equivalently $f \wedge a = f$.

These checks can be found for many other gate functions and are easily implemented based on truth tables or BDDs. If a check finds that some node a is not a valid candidate, all combinations in which a is present can be discarded. As rules with two operands we implemented AND-resub ($f \wedge a = f$) and OR-resub ($f \vee a = f$) according to Theorem 1. We also added XOR-resub, which is not canalizing and therefore cannot be concluded from checking a single operand only.[1]

Typically candidates are iterated in nested for loops. Therefore it is most effective to filter candidates based on nodes in the outer loops. However, also with several operands the technique can be applied in several steps as illustrated by the following example.

Example 2: If we plan to express a function f in terms of three nodes with functions a, b, and c using $f = a \vee (b \wedge c)$, we can first check whether a is valid. If a is invalid, one can continue. Otherwise, one can check b's validity to filter out candidates for c. When checking b, one can assume that $a = 0$, since otherwise the equation is satisfied independent of b. Therefore, one can reapply Theorem 1 to $f_{a=0} = b \wedge c$, which is $0/0$-canalizing in b.

Based on the last example, we add further resubstitution techniques using three operands, which are AND-OR resub ($f = a \vee (b \wedge c)$), OR-AND-resub ($f = a \wedge (b \vee c)$), but also XOR-AND-resub ($f = a \wedge (b \oplus c)$) and XOR-OR-resub ($f = a \vee (b \oplus c)$), in which both a can be filtered using Theorem 1. We also consider a MUX resub ($f = (a \wedge b) \vee (\bar{a} \wedge c)$). Disallowing a MUX with a constant data input, one can first filter the select input a by noting it is valid, if $(fa \neq 0) \wedge (f\bar{a} \neq 0)$. The next operands can be filtered using Theorem 1.

The proposed Boolean filtering allows logic synthesis to spend runtime only on profitable transformations, skipping unfruitful attempts. For example, considering the *voter* benchmark of the EPFL suite [5], ABC command *resub -K 10 -N 1* [4] takes about 0.09 seconds and reduces the size by about

[1] There are other more technical tricks that can help to speed up XOR resubstitution, but go beyond the scope of this paper.

14%. Increasing *N* to 2 and 3, which corresponds to 2 and 3 resubstitution, improves the size savings to 16% and 20% for 0.13 and 0.21 seconds spent, respectively. On the other hand, our proposed resubstitution technique with Boolean filtering, using the same cut size of 10, is able to reduce the original size by 26% in 0.1 seconds. Our size improvement is measured after AIG strashing, which decomposes our complex resub types into 2-input ANDs, for the sake of fair comparison with ABC's *resub* command.

B. Weaker Notion of MSPF

Recall that the MSPF of a node in a logic network is the set of all its permissible functions. The computation of the MSPF can be very time-consuming, especially when a logic network has many reconvergent paths. In addition, the MSPFs of all the gates and connections in a circuit have to be recomputed each time the circuit is transformed and reduced [13], [14].

To exploit the concept of permissible functions with affordable runtime, we present a weaker notion of MSPF, that we call *forward functional flexibility* (FFF) of the logic network w.r.t. a node n. FFF considers only the permissible functions generated by the *forward propagation* of a node's functionality and don't cares to its *transitive fan-out* (TFO). In contrast, the full MSPF considers *all* possible permissible functions by definition, thus also the ones originated by the interaction of a node with its *transitive fan-in* (TFI) and the rest of the network. Our experience indicates that FFF grasps a good amount of MSPF opportunities but still fits in a tight runtime budget [18].

The FFF(C, n) is computed as a second truth table[2] for a specific node n in a small or medium network C. The FFF(C, n) information can be interpreted as follows: where its entry is '1', the corresponding entry in the real truth table of n can be flipped without effecting the value at any of the primary outputs in C. This information introduces flexibility to Boolean methods, as it is sufficient to compare the nodes' functions only at their non-flexible input combinations; as demonstrated in the following example.

Example 3: Let n_1 and n_2 be two nodes in a network C, representing the functions f_1 and f_2, respectively. Then we can consider n_1 and n_2 equal in C, if

$$\big(f_1 \,\&\, \overline{\text{FFF}}(C, n_1)\big) = \big(f_2 \,\&\, \overline{\text{FFF}}(C, n_2)\big)\,.$$

Further, the previously presented filtering techniques can be enhanced by FFF. Reconsider Example 1: The validity check for node a is stronger when incorporating functional flexibility:

$$\big(f \,\&\, \overline{\text{FFF}}(C, f)\big) \rightarrow \big(a \,\&\, \overline{\text{FFF}}(C, a)\big)$$

Algorithm 1 shows the high-level procedure to compute FFF(C, n). The procedure starts by assigning maximum flexibility to the global FFF(C, n) (logic constant 1). The local don't care (DC) truth table for n is also set to logic constant 1. In the context of Algorithm 1, the *don't-care truth-table* DC(n, m) for some node m w.r.t. n has a different meaning than the FFF. In order to explain this, let us focus on the i^{th} entry of each table. If DC$(n, m)[i]$ is '0', it means that the m's truth table at index i is not sensible (*"does not care"*) to flipping the bit at index i of n's truth table propagated through n's TFO.

[2]The FFF computation procedure can be extended to use BDDs, or any other canonical representation, in place of truth tables.

Input : Logic network C, Current node n
Output: Forward Functional Flexibility FFF(C, n)

```
1  DC(n) = logic constant 1;
2  FFF(C, n) = logic constant 1;
3  foreach node m ∈ TFO(n) in topological order do
4      foreach fanin k of m do
5          i = 0;
6          if k has DC info then
7              DC-in(i) = DC(n, k);
8          else
9              DC-in(i) = logic constant 0;
10         i++;
11     end
12     SOP = get-sop-representation(m);
13     DC(n, m) = logic constant 0;
14     acc-sop = logic constant 0;
15     foreach term of the SOP do
16         term = logic constant 1;
17         flex-term = logic constant 0;
18         foreach literal of the term do
19             flex-term = dc-and(literal,
               DC-in(literal-index), term, flex-term);
20             term = literal & term;
21         end
22         DC(n, m) = dc-or(term, flex-term, acc-sop,
           DC(n, m));
23         acc-sop = acc-sop | term;
24     end
25     if m ∈ PrimaryOutputs(C) then
26         FFF(C, n) = FFF(C, n) & ¬DC(n, m);
27 end
28 return FFF(C, n);
```

Algorithm 1: Functional flexibility computation

In Alg. 1, the DC truth-table is propagated from n to the primary outputs in topological order. For each node, the DC truth-table is computed by processing the node's *sum-of-products* (SOP) representations. Note that while parsing the SOP, and/or operators are replaced by special operators (Alg. 2 and Alg. 3) that take into consideration the local don't cares.

Input : Truth tables (a, dca, b, dcb)
Output: ***dc-and***(a, dca, b, dcb)

```
1 aux1 = (a | dca) & dcb;
2 aux2 = (b | dcb) & dca;
3 res = aux1 | aux2;
4 return res;
```

Algorithm 2: ***dc-and***: and-2 function including don't cares for each operand

Input : Truth tables (a, dca, b, dcb)
Output: ***dc-or***(a, dca, b, dcb)

```
1 aux1 = (! a & ! dca) & dcb;
2 aux2 = (! b & ! dcb) & dca;
3 res = aux1 | aux2;
4 return res;
```

Algorithm 3: ***dc-or***: or-2 function including don't cares for each operand

When a primary output m is found during the procedure, FFF(C, n) is updated by and-ing itself with the complement of DC(n, m). It can be seen that DC(n, m) and FFF(C, n) have complementary meanings. Moreover, only the *common intersection* between the FFF of all outputs can be safely used for optimization purposes, which explains the and-ing. Once all the primary outputs have been processed, the final *forward functional flexibility* FFF(C, n) is returned by Algorithm 1.

IV. Global Boolean Resynthesis Flow

This section presents new Boolean optimization techniques, based on the theoretical and practical improvements described so far, forming altogether a novel Boolean resynthesis flow.

A. Resub for Complex Gates

We propose an enhanced resubstitution algorithm, capable of inferring complex gates, mapped or unmapped, with efficient runtime. Alg. 4 depicts the pseudocode. The top

Input : Logic network, cut-size, filter-volume, nresub (nrsb), zero-gain (zg), max-candidates (mc)
Output: Resynthesized logic network

```
1  list = topological-sort-network(network);
2  foreach node m in list do
3  |  if node is not a MFFC root then
4  |  |  continue;
5  |  cut = find-reconvergent-cut(m, cut-size);
6  |  expand-cut-into-window(cut);
7  |  if volume cut / size cut < filter-volume then
8  |  |  continue;
9  |  compute-truth-tables(window);
10 |  wdw = topological-sort-network(window);
11 |  foreach node n in wdw do
12 |  |  if (nrsb > -1) && zero-resub(n, window) then
13 |  |  |  continue;
14 |  |  if (nrsb > 0) && and-resub(n, wdw, zg, mc) then
15 |  |  |  continue;
16 |  |  if (nrsb > 1) && xor-resub(n, wdw, zg, mc) then
17 |  |  |  continue;
18 |  |  if (nrsb > 2) && ao-resub(n, wdw, zg, mc) then
19 |  |  |  continue;
20 |  |  if (nrsb > 3) && xa-resub(n, wdw, zg, mc) then
21 |  |  |  continue;
22 |  |  if (nrsb > 4) && ax-resub(n, wdw, zg, mc) then
23 |  |  |  continue;
24 |  |  if (nrsb > 5) && mux-resub(n, wdw, zg, mc) then
25 |  |  |  continue;
26 |  |  if (nrsb > 6) && mx-resub(n, wdw, zg, mc) then
27 |  |  |  continue;
28 |  end
29 end
30 network-cleanup-and-sweeping(network);
```

Algorithm 4: Resub for complex gates

procedure spans through the entire logic network, considering each node in topological order, but resubstitution is only applied to small/medium windows created around a node. The window computation proceeds as follows. First a reconvergent cut is found for the current node. Then, the cut is expanded w.r.t. its boundaries, i.e., every external node which is not contained by the cut, but has fanins inside the cut, is added. This process is continued until (i) no more nodes can be added or (ii) a volume limit is hit. The final result is a window, with as many inputs as leaves in the original cut but more outputs, due to the expansion. Resubstitution is then applied to this window. Windows that are too thin, i.e., with volume over input size ratio too small, are filtered because they are unlikely to lead to any advantageous resubstitution. Only windows passing the filtering tests move to truth table computation. Truth tables, functional support, and related properties, are calculated for every node in the window, using efficient bitwise manipulation as, e.g., discussed in [19]. At this point, each node of the window is tried for various resubstitutions, in a *waterfall model*, i.e., the first resubstitution succeeding is the one kept. The variable *nresub* determines how many different types of resub are tried, and thus controls the computational complexity of the algorithm. Without loss of generality, we assume *nresub* is set to *max-int* so all types of resub are tried. First, zero-resub is tried, where only equivalent gates in the window are merged, up to complementation. Functional support information can drastically speedup zero-resub: only candidates with exactly the same support as n are tried. If zero-resub returns 1, i.e., was successful, no other resubs are attempted for the same node and the loop moves to the next node in the window. Otherwise, other types of resub are tried. For each of these other resub, the size of the MFFC rooted at n is checked: if {extra-resub-nodes $\geq$ MFFC-size + zero-gain} the loop moves directly to the next node, because no size saving is possible. For example, and-resub introduces 1 extra node and the MFFC-size of n needs to be at least 2 to have advantageous resub. This costing can be extended to reduce the number of levels. More importantly, this costing can be made more accurate if the logic network is mapped, so real area savings can be measured in place of node savings. Inside each of the resub type, the filtering rules of Section III-A are used to evaluate only candidates leading to valid resubstitutions. Also, a maximum number of candidates is still used, even though rarely hit, to keep runtime under control for corner cases. The types of supported resub, in increasing computational complexity order, are: *and* (and-2 node/gate), *xor* (xor-2 node/gate), *ao* (and-or nodes/gates), *xa* (xor-and nodes/gates), *ax* (and-xor nodes/gates), *mux* (mux node/gate) and *mx* (mux-xor nodes/gates). If the regular polarity resub is not immediately successful, complementation at inputs and output of each resub is also tried. Finally, dead nodes cleanup and logic network sweeping is run.

B. Resub with weaker MSPF

The *forward functional flexibility* information introduced in Section III-B is particularly useful in the resub environment. It can be used to update the truth table information for the current node n in the window, so successive resub moves can take advantage of the don't cares flexibility. Alg. 5 shows the procedure for resub with FFF. Most of the procedure remains identical to Alg. 4, but with updates inside the window processing. For each node n evaluated inside the window, the FFF information is computed and the truth table for n is updated. If the successive resub is successful, no other nodes are tried and we skip to next window. This is necessary because the truth tables may be modified by the use of don't cares, so successive resub moves are potentially incorrect and FFF properties

Input : Logic network, cut-size, filter-volume, nresub, zero-gain, max-candidates
Output: Resynthesized logic network using FFF information

```
1  list = topological-sort-network(network);
2  foreach node m in list do
3      if node is not a MFFC root then
4          continue;
5      cut = find-reconvergent-cut(m, cut-size);
6      expand-cut-into-window(cut);
7      if volume cut / size cut < filter-volume then
8          continue;
9      compute-truth-tables(window);
10     wdw = topological-sort-network(window);
11     foreach node n in wdw do
12         FFF = compute-FFF(n, wdw);
13         update-truth-table-with-flexibility(n, FF);
14         run-resub-flow();
15         if resub with FFF is successful then
16             break;
17     end
18 end
19 network-cleanup-and-sweeping(network);
```

Algorithm 5: Resub with *forward functional flexibility*

invalid. We also evaluated alternative approaches to window-skipping, where FFF and truth tables are incrementally updated after a successful resub. We experimentally found that window-skipping is still more advantageous (QoR/runtime-wise) because (i) permits exploring diverse cones of logic and (ii) covers more don't cares combinations.

C. Refactoring into Multiple Representation Forms

Refactoring aims at re-expressing a cut, or a window, with a new, potentially more compact, logic structure. The change is accepted if there is an improvement in the target cost metric (area, delay, etc.). Refactoring differs from resubstitution in the scope of the structural change: while resub changes one node at a time, refactoring operates on a large cone of logic at once. Compared to rewriting, refactoring supports larger cuts and can operate on multi-output functions (windows).

Standard refactoring is based on SOP representation, and thus uses traditional collapsing, minimization and refactoring algorithms [20]. As many new representation forms emerge in EDA [21], and offer advantages over SOP, we developed a generalized refactoring engine which can support generic data structures & optimization techniques. We call it *window export & import* (WEI) package. WEI is applied node-wise, and works as follows. For the given node n, it computes a cut using reconvergence driven methods. The cut can be extended into a window, using a similar approach to the one described for resub. Then, a new logic network is created, named *weiNet*, which duplicates the logic of the window. Each leaf of the window/cut becomes a *primary input* (PI) of *weiNet*. Each node of the window, which is not a PI, and has at least one fanout outside of the window itself becomes a *primary output* (PO) of *weiNet*. At this point, a generic synthesis technique can be applied to the newly formed network. After synthesis, the new network characteristics are measured. If an improvement is seen in the chosen metric, the new network is imported back to the original network. Because of the way *weiNet* is built, sharing of the logic below each POs is enforced and maximized. Thus, no extra duplication happens when importing back the new logic network.

Some beneficial synthesis techniques that we have experimented with the WEI engine, over small/medium *weiNet* sizes, are: SOP collapsing and factoring, ESOP collapsing and XOR-decomposition, BDD collapsing and dominator-based decomposition, BBDD restructuring for XORs, SPP collapsing and minimization, and others.

In summary, WEI enables high-effort optimization methods to run on small/medium sub-networks, with guaranteed maximal sharing of nodes and contained computational complexity.

D. Integration into a Global (Re)Synthesis Flow

We have integrated the optimization techniques presented so far in an industrial logic optimization engine, together with well known state-of-the-art methods. Then, we created a Boolean resynthesis script, consisting of the following commands:

```
rw; rs -c 9; rs -c 10; rfs -c 9;
rwz; mf -c 9; rsz -c 10; rw;
rfb -c 12; rs -c 9; rfs -c 10;
rs -c 10; rw; rs -c 12; rsz -c 12;
rwz; rfs -c 10; mf -c 10;
rs -c 12; rfbz -c 14; mf -c 11; rwz
```

Where *rw* is state-of-the-art rewriting and balancing [7], [22], *rs* is resub for complex gates, *mf* is resub with FFF information, *rfs* is generalized refactoring based on SOPs and *rfb* is generalized refactoring based on BDDs. Commands ending with *z* stands for *"accept zero gain transformations"*. The extra switch *-c* denotes the maximum cut/window size. By default all available types of resub are tried, with maximum number of candidates equal to 100. Faster versions of this script only look for a subset of resub types, e.g., *nresub = 3* for *rs* and *mf* commands. All commands can be programmed to work either on unmapped, e.g., AIG, or mapped logic networks. The Boolean resynthesis script produces improved results when iterated multiple times, e.g., 2-5 times, depending on the specific runtime budget. More iterations need more diverse optimization methods to be intertwined with Boolean resynthesis, to escape deeper local minima.

V. Experimental Results

In this section, we evaluate the efficacy of our proposed Boolean resynthesis flow. First, we challenge Boolean resynthesis to improve the best results in the EPFL benchmark suite [5]. Finally, we integrate Boolean resynthesis in an industrial EDA flow, and show sensible QoR gains post place & route.

A. Methodology

We implemented Boolean resynthesis as part of a commercial design automation solution. In the EDA flow, Boolean resynthesis runs after the initial logic structuring, which mainly aims at reducing area. So, Boolean resynthesis targets size reduction in the logic network. Tight control on the number of levels and the number of nets is enforced during Boolean resynthesis: this is known to correlate with delay and congestion later on in the flow.

We also implemented Boolean resynthesis as a standalone optimization package, to run tests on academic benchmarks.

B. EPFL Benchmarks

The EPFL benchmark suite project keeps track of the best synthesis results, mapped into LUT-6, generated by EDA research groups. In this work, we challenge the area category of the EPFL suite. As the EPFL best results come mapped into LUT-6, we use Boolean resynthesis in mapped mode to improve on top of them. New gates introduced by Boolean resynthesis have less than 6 inputs, so the output of our Boolean resynthesis is already compliant with the EPFL competition rules. In order to form meaningful windows of large-fanin nodes, we increased the cut size of the Boolean resynthesis script, adding *+6* to each command. The runtime of Boolean resynthesis still remained comparable to traditional restructuring/mapping scripts for the largest benchmarks processed. Our results are summarized by Table I. Thanks to Boolean resynthesis, we improved 10 of the previous best size (area) results. Level count improved for the hypotenuse benchmark, as highlighted in green color. For the other benchmarks, level count is not increased.

TABLE I
NEW BEST AREA RESULTS FOR THE EPFL SUITE

Benchmark	I/O	LUT-6 count	Level Count.
arbiter	256/129	**403**	23
i2c	147/142	**211**	7
log2	32/32	**6570**	119
mem_ctrl	1204/1231	**2117**	22
priority	128/8	**108**	26
sin	24/25	**1228**	55
hypotenuse	256/128	**40385**	**4527**
voter	1001/1	**1297**	17
sqrt	128/64	**3076**	1106
square	64/128	**3243**	74

Our circuit implementations can be downloaded at [23].

C. ASIC Results

We tested a commercial EDA flow, empowered with our Boolean resynthesis, on 50 state-of-the-art ASICs, coming from major electronics industries. We cannot provide details on each ASIC benchmark because of non-disclosure agreements. However, we present average results w.r.t. a baseline flow without our Boolean resynthesis techniques. The post place & route results are summarized by Table II.

TABLE II
POST PLACE&ROUTE RESULTS ON 50 INDUSTRIAL DESIGN

Flow	Area	Leakage	WNS	TNS	Runtime
Baseline	1	1	1	1	1
Proposed flow	**-2.67%**	**-2.11%**	**-0.44%**	**-5.48%**	+1.35%

Our complete design flow, embedding Boolean resynthesis, enables sensible area & leakage reductions, -2.67% and -2.11% respectively, on average, and also good WNS/TNS improvements, at only $+1.35\%$ runtime cost.

To fully appreciate the impact of our techniques, consider that the relative area reduction is calculated on the whole chip. This also includes sequential elements, IPs and other blocks where Boolean resynthesis does not directly operate.

VI. CONCLUSIONS

In this paper, we revisited fundamental data structures and algorithms for Boolean resynthesis, with the aim to find more optimization opportunities at affordable runtime cost. The major contributions of this work are: (i) a theory of Boolean filtering, to drastically reduce the number of gates processed and still retain all possible optimization opportunities, (ii) a weaker notion of MSPF, which can be computed efficiently via truth tables, (iii) a generalized refactoring engine and (iv) a practical Boolean resynthesis flow, which combines the proposed techniques. Using our Boolean resynthesis on LUT-6 mapped networks, we improved 10 of the best known area results in the EPFL synthesis competition. Embedded in a commercial EDA flow for ASICs, our Boolean resynthesis flow reduced the area by -2.11%, and total negative slack by -5.48%, after physical implementation, at small runtime cost.

REFERENCES

[1] 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.

[2] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[3] A. Mishchenko, R. K. Brayton, J. R. Jiang, and S. Jang, "Scalable don't-care-based logic optimization and resynthesis," *ACM Trans. on Reconfigurable Technology and Systems*, vol. 4, no. 4, pp. 34:1–34:23, 2011.

[4] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Computer Aided Verification*, 2010, pp. 24–40.

[5] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *Int'l Workshop on Logic and Synthesis*, 2015.

[6] S. P. Khatri and K. Gulati, Eds., *Advanced Techniques in Logic Synthesis, Optimizations and Applications*. Springer, 2011.

[7] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "DAG-aware AIG rewriting a fresh look at combinational logic synthesis," in *Design Automation Conference*, 2006, pp. 532–535.

[8] A. Mishchenko and R. K. Brayton, "Scalable logic synthesis using a simple circuit structure," in *Int'l Workshop on Logic and Synthesis*, 2006, pp. 15–22.

[9] V. N. Kravets and P. Kudva, "Implicit enumeration of structural changes in circuit optimization," in *Design Automation Conference*, 2004, pp. 438–441.

[10] A. Mishchenko, J. S. Zhang, S. Sinha, J. R. Burch, R. K. Brayton, and M. Chrzanowska-Jeske, "Using simulation and satisfiability to compute flexibilities in Boolean networks," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 25, no. 5, pp. 743–755, 2006.

[11] M. Soeken, L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Exact synthesis of majority-inverter graphs and its applications," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2017, accepted.

[12] W. Haaswijk, M. Soeken, L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "A novel basis for logic optimization," in *Asia and South Pacific Design Automation Conference*, 2017.

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

[14] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney, "The transduction method-design of logic networks based on permissible functions," *IEEE Trans. on Computers*, vol. 38, no. 10, pp. 1404–1424, 1989.

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

[16] S. Kauffman, "The large scale structure and dynamics of gene control circuits: An ensemble approach," *Journal of Theoretical Biology*, vol. 44, no. 1, pp. 167–190, 1974.

[17] D. E. Knuth, *The Art of Computer Programming, Volume 4A*. Addison-Wesley, 2011.

[18] L. G. Amarù, P. Vuillod, J. Luo, and J. Olson, "Logic optimization and synthesis: Trends and directions in industry," in *Design, Automation and Test in Europe*, 2017, pp. 1303–1305.

[19] H. S. Warren, Jr., *Hacker's Delight*. Addison-Wesley, 2002.

[20] R. K. Brayton and C. T. McMullen, "The decomposition and factorization of Boolean expressions," in *Int'l Symp. on Circuits and Systems*, 1982, pp. 49–54.

[21] L. G. Amarù, *New Data Structures and Algorithms for Logic Synthesis and Verification*. Springer, 2017.

[22] A. Mishchenko, R. K. Brayton, S. Jang, and V. N. Kravets, "Delay optimization using SOP balancing," in *Int'l Conf. on Computer-Aided Design*, 2011, pp. 375–382.

[23] Https://lsi.epfl.ch/benchmarks.