Scalable Boolean Methods in a Modern Synthesis Flow

Eleonora Testa[1,2], Luca Amarú[1], Mathias Soeken[2], Alan Mishchenko[3], Patrick Vuillod[1],
Jiong Luo[1], Christopher Casares[1], Pierre-Emmanuel Gaillardon[4], Giovanni De Micheli[2]

[1]Synopsys Inc., Design Group, Sunnyvale, California, USA
[2]Integrated Systems Laboratory, EPFL, Switzerland
[3]Department of EECS, UC Berkeley, Berkeley, California, USA
[4]LNIS, University of Utah, Salt Lake City, Utah, USA

***Abstract*—With the continuous push to improve *Quality of Results* (QoR) in EDA, Boolean methods in logic synthesis have been recently drawing the attention of researchers. Boolean methods achieve better QoR than algebraic methods but require higher computational cost. In this paper, we introduce the *Scalable Boolean Method* (SBM) framework. The SBM consists of 4 optimization engines designed to be scalable in a modern synthesis flow. The first presented engine is a generalized resubstitution framework based on computing, and implementing, the Boolean difference between two nodes. The second consists of a gradient-based AIG optimization, while the third one is based on heterogeneous elimination for kerneling. The last proposed engine is a revisiting of maximum set of permissible functions computation with BDDs. Altogether, the SBM framework enables significant synthesis results. We improve 12 of the best known area results in the EPFL synthesis competition. Embedded in a commercial EDA flow, the new Boolean methods enable -2.20% combinational area savings and -5.99% total negative slack reduction, after physical implementation, at contained runtime cost.**

I. Introduction

As transistor scaling slows down at advanced technology nodes, e.g., 10 nm, 8 nm, 7 nm and beyond, EDA innovations are becoming essential to keep up with the (expected) *Quality of Results* (QoR). This motivates EDA researchers to revisit high-quality and high-computational-complexity optimization methods in light of modern computing capabilities. For instance, the recent work in [1] showed improvements to Boolean resynthesis, enabling some high-quality Boolean methods to be runtime affordable. In this paper, we extend the work from [1] and introduce the *Scalable Boolean Method* (SBM) framework. The SBM presents a new set of Boolean methods, orthogonal to the existing ones, capable of finding undiscovered optimization tradeoffs, while remaining scalable in a modern synthesis flow.

The main contributions of this paper, which all together make SBM efficient, are:

1) a novel Boolean resubstitution framework which optimizes logic networks by computing and implementing the Boolean difference between two nodes,

2) a gradient-based *And-Inverter Graphs* (AIGs) optimization engine which learns the most effective AIG transformations during the optimization,

3) heterogeneous elimination and kerneling to enhance division and logic sharing to work on heterogeneous thresholds within the same network,

4) a revisiting of *Maximum Set of Permissible Functions* (MSPF) computation using *Binary Decision Diagrams* (BDDs).

Altogether, the SBM optimization framework enables significant synthesis results. We show substantial improvements over the smallest known AIGs for EPFL benchmarks [2]. For example, we show 1.5$\times$ size reduction in the smallest known AIG for the EPFL *arbiter* benchmark. By mapping onto LUT-6 the AIGs obtained through our Boolean methods, we improve 12 of the best known area results in the EPFL synthesis competition [2]. Embedded in a commercial EDA flow for ASICs, the SBM framework enables -2.20% combinational area savings and -5.99% total negative slack reduction, after physical implementation, at contained runtime cost.

The remainder of this paper is organized as follows. Section II provides some background on Boolean methods in synthesis and discusses the motivation for this work. Section III proposes a generalized resubstitution framework based on Boolean difference optimization. Section IV introduces the remaining optimization techniques: gradient-based AIG optimization, heterogeneous elimination and kerneling, and MSPF computation with BDDs. Section V shows experimental results for the SBM framework over academic benchmarks and commercial ASIC designs. Section VI concludes the paper.

II. Background and Motivation

A. Boolean Logic Optimization

Approaches to logic network optimization are divided into algebraic methods and Boolean methods. While algebraic methods are faster, Boolean methods achieve better results [3]. Boolean transformations rely on a complete Boolean algebra and functional properties of logic circuits, which often include don't cares conditions. *Permissible functions* are one of the many examples of don't cares interpretation in synthesis. If the function at a node n may be changed to another function without changing the behavior at the primary outputs, then the new function is called a permissible function for node n [4]. The set of all permissible functions for a node n is called its *Maximum Set of Permissible Functions* (MSPF). As of today, different logic reasoning engines are available for gathering functional properties of a logic circuit. Here, we give some background on truth tables, BDDs and SAT, as they are used as engines in the SBM framework.

A *truth table* is a canonical representation of a Boolean function where the function values are listed for all input combinations. When Boolean methods are applied to small

978-3-9819263-2-3/DATE19/©2019 EDAA

windows of logic (≈ 15 inputs), they enable fast computation and equivalence checking of two functions. They are usually used together with partitioning techniques to allow Boolean optimization. As an example, the Boolean resynthesis flow in [1] uses truth tables as reasoning engine to compute MSPF.

Binary Decision Diagrams (BDDs, [5]) are directed acyclic graphs representing a Boolean function. Each internal node of the BDD implements the Shannon expansion $f = x_i f_{x_i} \oplus \bar{x}_i f_{\bar{x}_i}$ of the function with respect to a variable x_i, where f_{x_i} and $f_{\bar{x}_i}$ are the two *cofactors*. BDDs are largely employed in Boolean optimization methods [3], [6]. Like truth tables, BDDs can be used to check if a function is a permissible replacement of another (≈ 20 inputs functions). This is usually performed by checking functional equivalence, at either local or global level [3]. BDDs are also employed for representing and minimizing Boolean relations [6]. Boolean relations are considered a superior version of don't cares [3], used to capture the flexibility of multi-output circuits. Further, BDDs are also used for logic function decomposition. As an example, BDS [7] is an optimization system for the synthesis of AND/OR and XOR-based functions using BDDs.

SAT solvers have recently been used as Boolean method engine for don't cares computation. A SAT problem takes a formula representing a Boolean function and decides if there is an assignment of the variables for which the function is equal to 1 (satisfiable). The work in [8] presents a method to cast don't cares computation as a SAT problem. More recently, a SAT-based redundacy removal approach has been presented [9].

The engines presented so far can be used for verifying the applicability of Boolean transformations. An example of a Boolean transformation, which will be used in the following discussion, is *resubstitution*. Resubstitution rewrites the function of a node n as a new function of other nodes already present in the network. If the new implementation of the node is more compact than the previous one, resubstitution results in area savings. We refer the interested reader to [3], [10], [11] for a more detailed review on Boolean methods.

B. Motivation

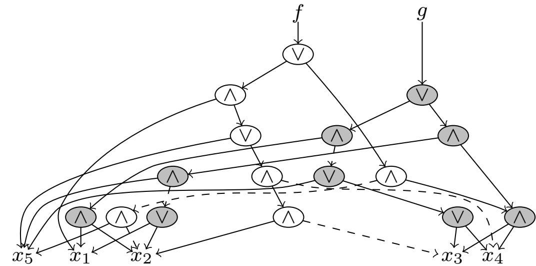
Boolean optimization methods are more powerful and complete than algebraic methods but come at a higher runtime cost. As a consequence, their applicability in automated design flows is limited, thus leaving possible optimization opportunities unexplored. In this paper, we present a novel Boolean optimization framework, called SBM. We specifically design it to be scalable and to unveil further optimization opportunities in modern synthesis flows.

III. Boolean Difference Based Optimization

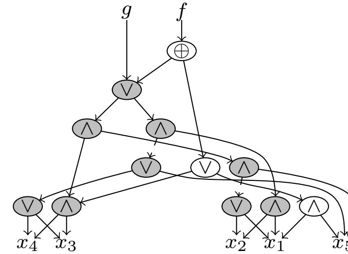
This section presents a novel Boolean resubstitution framework based on Boolean difference computation and implementation.

A. Theory

The *Boolean difference* of two Boolean functions $f(x_1, \ldots, x_n)$ and $g(x_1, \ldots, x_n)$ is defined as [3]: $\frac{\partial f}{\partial g} = f \oplus g$, where $\oplus$ is the XOR operator. The equation indicates whether the two functions are functionally equivalent (i.e., the difference value with respect to inputs assignments is 0) or not (i.e., they have difference equal to 1).



(a) Logic network for functions f and g (in gray)



(b) Function f rewritten as $f = \frac{\partial f}{\partial g} \oplus g$

Fig. 1: Boolean difference example

In this paper, we take advantage of the Boolean difference to build a resubstitution framework. In the following discussion, f and g are used both for the corresponding nodes in the logic network and for the function they represent. Each function f can be written as $f = \frac{\partial f}{\partial g} \oplus g$. While g is a node already existing in the logic network, the term $\frac{\partial f}{\partial g}$ needs to be retrieved in a compact logic form, so it could lead to size/depth minimization. Consider, as an example, the logic network for function f and g in Fig. 1(a). Each node in Fig. 1(a) is a 2-input gate, and dashed edges represent inverters. The total number of nodes is the *size* of the network, and the number of levels is its *depth*. The function g is the one highlighted in gray in both Fig. 1(a) and (b), while function f is the one written as $\frac{\partial f}{\partial g} \oplus g$ in Fig. 1(b). Due the small size of the Boolean difference network, the total number of nodes is reduced.

In this work, we exploit the concept of Boolean difference in logic optimization. We focus on size reducing transformations, but depth reducing techniques could be developed in a similar manner. We refer to function f and g as *candidates* for Boolean difference, and to the inputs $x_1, \ldots, x_n$ as their *support*. First, we discuss how to select the two candidates f and g, then, we present an algorithm to compute and implement the Boolean difference. Finally, we present the global synthesis flow.

B. Identifying Viable Candidates

To ensure the scalability of this Boolean method, we evaluate and apply the Boolean difference locally on limited size *circuit partitions*. The partitions are created by collecting all the nodes in topological order and by sorting them according to the similarity of their structural support. Each partition respects some predefined characteristic, e.g., maximum number of primary inputs, maximum number of internal nodes n, maximum number of levels, etc. In our implementation, we give priority to the limit on the maximum number of levels, as they correlate

Algorithm 1 Boolean difference computation and implementation using BDDs

```
Input: Two nodes f and g, xor_cost, all_bdds
Output: A new node boolean_diff equal to ∂f/∂g ⊕ g
 1: boolean_diff ← 0
 2: bddf ← all_bdds(f)
 3: bddg ← all_bdds(g)
 4: bdd_diff ← bddf ⊕ bddg
 5: if bdd_diff already exists in all_bdds() then
 6:     return corresponding node
 7: end if
 8: if (size(bdd_diff) > threshold) then
 9:     return null
10: end if
11: saving ← mffc(f) + nodes_sharing
12: if (size(bdd_diff) + xor_cost > saving) then
13:     return null
14: end if
15: bdiff_node ← bdd_to_node(bdd_diff)
16: boolean_diff ← bdiff_node ⊕ g
17: return boolean_diff
```

Algorithm 2 Resubstitution flow based on Boolean difference

```
Input: Network N, xor_cost
Output: Optimized network.
 1: lists ← topological_sorted_partitions(N)
 2: for each list in lists do
 3:     all_bdds ← BDDs for all nodes in list
 4:     for nodes f in list do
 5:         for nodes g in list do
 6:             if f = g then
 7:                 continue
 8:             end if
 9:             if f and g are not good candidates then
10:                 continue
11:             end if
12:             diff ← Boolean_difference(f, g, xor_cost, all_bdds)
13:             if size(diff) <= size(f) then
14:                 Change f with diff in N
15:             end if
16:         end for
17:     end for
18: end for
19: return N
```

with the complexity of the reasoning engine selected for the Boolean difference computation. Nevertheless, we also ensure partitions to have limited size and limited number of primary inputs. Experimentally, we found promising bounds on the number of levels ranging from 5 to 30, resulting in partitions with controlled maximum size of 1000 nodes.

In order to find good candidates f and g, all pairs of nodes inside each partition are considered. The supports for the computation are the primary inputs of the partition itself. As this requires the evaluation of n pairs of nodes for each node, in the worst case, the time complexity of the resubstitution framework is quadratic w.r.t. the partition size n. Experimentally, to reduce the time complexity, we fixed the maximum number m of pairs to be tried. Structural filtering can also accelerate the computation. For example, the algorithm does not consider nodes with less than one element in their shared support, and it also neglects cases where f is completely included in g, or partially included up to a certain threshold. Functional filtering similar to the one in [1] also helps speeding up the computation. After all speed ups, we can apply the method to EPFL *i2c* and *cavlc* benchmarks monolithically, with a runtime of 2.3 and 1.2 seconds, respectively.

C. Computing and Implementing The Difference

BDDs are the selected data structure to compute and implement the Boolean difference. The pseudocode is depicted in Alg. 1. Recall that f and g are two nodes belonging to the same partition. The BDDs for all nodes in the partition are precomputed and stored in the hashtable *all_bdds*. The algorithm computes the BDD of the Boolean difference as XOR of the two BDDs. Thanks to the limited size of the partition, BDDs allow fast Boolean difference computation. If the BDD of the difference already exists in the hashtable, the corresponding node is returned. In our implementation, we did not perform any BDD variables ordering, as we are dealing with small BDDs. This saves runtime, but it requires a higher amount of memory to be used by the BDD package. The memory usage plays a critical role. For instance, for the EPFL *cavlc* benchmark, the algorithm does not converge in a reasonable amount of time unless the memory used for the BDD of the difference is freed at each iteration. In this last case, the algorithm was applied on the whole network, which has 10 inputs and more than 600 nodes. To further prevent memory issues, we set a maximum memory limit for the employed BDD package. The BDD computation is bailed out if the maximum memory limit is hit. This case results into a BDD of size 0 for the given node, which will be disregarded in the next steps of the algorithm.

Afterwards, structural filtering is applied on the BDD. In case the BDD does not meet the size requirements, Alg. 1 returns *null*, which means the current pair of nodes can be skipped. First, we limit the size of the BDD (lines 8–10 in Alg. 1) to consequently limit the size of the difference network once its BDD is merged into the AIG. This usually ensures a limited size implementation for the Boolean difference, but it may overlook some optimization opportunities. Empirically, we found 10 to be a suitable tradeoff to have good QoR and feasible runtime. The second filter skips pairs of nodes that could result in a larger network implementation. Experimentally, we skip nodes whose *saving* is smaller than the empirical threshold set by the BDD size and the xor_cost. The saving resulting from the Boolean difference is the sum of the size of the *Maximum Fan-out Free Cone* (MFFC, [12]) of f and the total sharing of nodes between the Boolean difference implementation and the existing network. The size of the BDD sets a lower bound on the number of AIG nodes to implement the Boolean difference. The xor_cost is the number of AIG nodes needed to implement the functionality of a two-input XOR. According to the specific technology involved, the XOR node has a different area ratio as compared to AND/OR nodes, so the xor_cost can have a different value.

The algorithm concludes with the implementation of the Boolean difference node (lines 15 in Alg. 1) as an AIG, obtained using *structural hashing* (strashing) on the corresponding BDD. Optimization algorithms from the state-of-the-art are applied on the AIG to guarantee an optimized implementation.

D. Global Resubstitution Flow

We integrate the candidates selection and the Boolean difference computation into a resubstitution framework. Alg. 2 depicts the pseudocode. The flow applies the resubstitution framework to each partition N of the entire network. The

partitions can be chosen to be distinct or overlapping to cover more optimization opportunities. The algorithm precomputes and stores all BDDs in the hashtable, and considers all nodes in topological order. Trivial pairs of nodes are skipped according to criteria discussed in Section III-B. Thanks to the use of BDDs, information needed for functional filtering of pairs are immediately available. Alg. 1 is used to achieve the new implementation of f using the Boolean difference. Alg 2 accepts a new implementation of f only if (i) it leads to size minimization, or (ii) it does not increase the number of nodes. This second case could reshape the network, open new optimization opportunities and help escaping local minima.

IV. Intelligent Optimization Engines

This section introduces the remaining methods of SBM, including gradient-based AIG optimization, heterogeneous elimination for kerneling, and MSPF computation with BDDs.

A. Gradient-Based AIG Minimization

AIG optimization traditionally consists of a predetermined sequence of primitive optimization techniques, forming a so-called *script*, which is homogeneously applied to the whole network [13]. One of the most popular AIG script in academia is *resyn2rs* from ABC [13], with major primitive techniques being *rewrite, refactor* and *resub*. In this work, we aim at making AIG optimization automatically *adaptive* and *diverse*. We foresee our tool to be adaptive by learning the most effective AIG transformations during the optimization script. This is achieved by using gradient computation of the gain, and it allows us to modify online the next attempted transformations accordingly. We aim at making our tool diverse by trying different types of AIG transformations on the same region of logic. This makes results compete locally rather than globally.

The gradient based AIG engine we propose runs together with a partitioning engine, either small scale or large scale depending on the intended scope of the optimization. We consider best result selection performed either in parallel or in a waterfall model. Within the waterfall model, the first successful move is picked, and all other moves are not tried. This leads to better runtime as compared to parallel model but it may overlook optimization opportunities. In the proposed AIG engine, the waterfall model is a good tradeoff between runtime and QoR. We define AIG optimization moves, which are primitive transformations applicable locally. We consider the following moves: *rewriting*, *refactoring*, *resub*, *mspf resub* and *eliminate, simplify & kerneling*. All moves other than *rewriting* are available in low and high effort modes, trading runtime for QoR. All moves have an associated cost, which depends on their runtime complexity. The optimization engine starts by trying unit cost moves for each partition, and by recording[1] the gain of the best one. Until $gain > 0$, cheap moves are iterated over the network and all its partitions. As we hit local minima ($gain = 0$), higher cost moves start to be introduced in the AIG engine. The most successful moves and their sequence are recorded during the optimization to allow moves with high success likelihood on the current design to be tried with higher priority in the next iterations. The gradient based AIG engine is called together with a *cost budget*, which determines how many moves can be tried. The budget can be automatically increased by the AIG engine, if the gain gradient exceeds a certain threshold over last k iterations. In other words, the AIG engine continues simplifying a logic network if the optimization trend is good enough, or terminates early if the gain gradient is 0 over the last k iterations.

In our experiments, we obtained the best AIG optimizations seen over academic and industrial benchmarks by using a cost budget equal to 100 and $k = 20$, with minimum gain gradient equal to 3%. In the experimental results section we will show some of the smallest AIG known for the EPFL competition [2], obtained through such AIG engine.

B. Heterogeneous Elimination for Kernel Extraction

Kernel extraction [10] is one of the most effective techniques in logic optimization. This is thanks to the fact that it allows us to share large portions of logic circuits, which are hard to find with other techniques. For example, kernel extraction is able to find common factors between very wide (hundreds to thousands of inputs) operators appearing in HDL descriptions of decoders and control logic.

The effectiveness of kernel extraction depends on the properties and characteristics of the nodes' SOPs. Indeed, prior to kernel extraction, node elimination[2] is often used to create larger SOPs. Elimination keeps under control the maximum number of terms or literals, and enables more extraction opportunities to be found. However, elimination is also usually run network-wise homogeneously, i.e., with the same thresholds on maximum number of terms or literals. In this way, the resulting SOPs have similar size but not similar characteristics, which is where the extraction opportunities arise.

In this work, we enhance *elimination - kernel extraction* to work on heterogeneous thresholds within the same network. While, in order to be exact, one would have to study the correlation between a logic circuit characteristic and the effectiveness of eliminate before kerneling, this appears to be an intractable problem. We take advantage of partitioning engines, whose computation can be distributed in parallel, to accomplish heterogeneous elimination - kernel extraction. The idea is to use different elimination thresholds depending on circuit characteristics. Even though kernel extraction is not a Boolean method, we categorized the *eliminate* enhancement as Boolean because it applies, more generally, to Boolean division as well.

We first partition the network, with given partition characteristics, and we apply elimination - kernel extraction to each partition with different eliminate thresholds. We only keep the best one, e.g., the one reducing the largest number of literals of the partition. The elimination process works as follows. We go over all nodes in the partition, and for each node, we estimate the variation in the number of literals in the partition that would result from the collapsing of the node into its fanouts. If this variation is less than the specified threshold, the collapsing is performed. The operation is repeated until no node gets collapsed. Empirically, we found useful to try the following eliminate thresholds: (-1, 2, 5, 20, 50, 100, 200, 300).

C. MSPF Computation with BDDs

The maximum set of permissible functions (MSPF) is one of the most powerful interpretations of don't cares for synthesis. The work in [1] proposed truth table methods to approximate MSPF during resubstitution. In this work, we

[1] All moves are designed to have $gain \geq 0$ at all times, otherwise the corresponding change is reverted.

[2] Node elimination, also known as forward node collapsing, aims at collapsing a node into its fanouts' SOPs. As a result, the node is eliminated.

propose a BDD-based version of MSPF logic optimization, which works on larger sub-circuits than those considered in [1].

The BDD-based MSPF optimization algorithm operates as follows. First, nodes are arranged in topological order, and further sorted w.r.t. an estimated saving metric for each node. The MSPF information is computed for each node via cofactoring. Specifically, the positive (negative) cofactor of the node w.r.t. each primary output is computed using BDDs [14], and stored as an array of BDD formulas, f_0 (f_1), with $size = |PO|$. At this point the $mspf(node)$ information is initialized to logic 1: $mspf(node) = bdd(1)$. Then, the actual computation loops over all POs, and updates the MSPF as: $mspf(node) = mspf(node) \wedge ((\bar{f}_0(po_i) \oplus f_1(po_i)) \vee dc(po_i))$, where po_i is the $i\text{-}th$ PO under consideration, and $dc(po_i)$ is any pre-existing don't care condition at the $i\text{-}th$ PO. The computation stops if, at any point of the loop, no MSPF is found for the current node, i.e., $mspf(node) = bdd(0)$. Otherwise, the MSPF information is passed to drive the successive optimization steps. Based on the permissible functions computed, the MSPF optimization algorithm can be reapplied on each fanin of the current node to reduce area, or another optimization metrics. For example, it is efficient to check via BDDs if changing a fanin of node still respects: $bdd(node_{new}) \wedge \neg mspf(node) = bdd(node_{old}) \wedge \neg mspf(node)$ In that case, the fanin is "connectable" as it generates a permissible function at the current node. From there, standard MSFP optimization algorithms [1] may be applied on top. As in the Boolean difference implementation with BDDs, in the MSFP algorithm we set a maximum memory limit. Also in this case, the algorithm sets the BDD size of the node to 0 if it hits the memory limit. The computation can then continue by considering the other nodes.

Another key enhancement to this technique, as compared to [1], is to look not just for one but for many connectable fanins under MSFP. Among all these, only an irredundant subset is actually tried. With truth tables, or even SAT, finding many connectable fanins would be a quite expensive task. With BDDs it is possible to perform such global queries more efficiently thanks to BDD strong canonicity, in modern packages, and efficient use of unique tables [15]. As a consequence, QoR improves with BDD-based MSPF computation because of the larger subset of solutions reachable.

V. Experimental Results

In this section, we evaluate the efficacy of the SBM framework for synthesis. First, we consider the EPFL logic synthesis competition [2]. In this scenario, we outperform previous EPFL best results coming from various research groups in industry and academia. Finally, we integrate our Boolean methods in an industrial EDA flow for ASICs, and show sensible QoR gains post place & route.

A. Methodology

We implemented the scalable Boolean methods as part of a commercial design automation solution. We target size reduction of logic networks, as, in the EDA flow, Boolean methods are frequently called during logic structuring, which mainly aims at reducing area. Nevertheless, we enforced a tight control on the number of levels and the number of nets during synthesis, as this is known to correlate with delay and congestion later on in the flow.

We have integrated all the optimization techniques presented so far in an industrial logic synthesis tool, together with state-of-the-art methods. We created a Boolean resynthesis script which runs the following optimizations:

- AIG optimization: this consists of both state-of-the-art methods [1] and our gradient-based AIG minimization,
- heterogeneous elimination for kernel extraction, applied on partitioned networks of medium-large sizes,
- enhanced MSPF computation, using partitions of medium size and BDDs,
- collapse and Boolean decomposition, applied on reconvergent MFFC of the logic network,
- Boolean difference-based optimization to unveil hard to find optimization and escape local minima,
- SAT-based sweeping and redundancy removal as in [9].

The optimization flow is iterated twice, with different *efforts*. Further, after each transformation, the logic network is translated into an AIG in order to have a consistent interface and costing between the various steps of the flow.

We also implemented the SBM framework as a standalone optimization package, to run tests on academic benchmarks.

B. EPFL Benchmarks

We present here our results on the 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 (i.e., number of LUTs) category of the EPFL suite [2]. As the EPFL best results come mapped into LUT-6, we use the ABC [13] command *"if -K 6 -a"* in order to map our AIGs. It is indeed known that LUT-6 minimization does not follow strictly AIG minimization. In order to make our techniques work in general for the LUT-6 experiment, we adapted our tool accordingly. We inserted selective strashing of LUTs, over previous best results, with optimization and remapping on smaller partitions, in order to preserve some of the good LUT-6 structures. Nevertheless, for some of the benchmarks (e.g., *arbiter*, *router*), the optimization script run on the original unoptimized AIG [2], followed by plain *"if -K 6 -a"*, was enough to beat previous best results. On the other hand, other benchmarks (e.g., *max*) are mostly improved by the Boolean difference method, which is capable of untangling reconvergent logic not touched by other techniques.

Our results are summarized in Table I. We improved 12 of the previous best size (area) results[3]. Our improvements range from just a few LUTs to several (tens) for large circuits. We improve both the size results presented in [1] and the ones coming from [16]. It is worth mentioning that the EPFL benchmarks have been optimized several times in the last 3 years by the most advanced techniques both from industry and academia. This makes each improvement (even if relatively small), significant. Our circuit implementations can be downloaded at [2].

As already discussed above, for some other benchmarks, a smaller AIG was not resulting in the best LUT-6 result. We report the smallest AIGs obtained with our optimization methodology in Table II. The size of the AIGs is smaller as compared to the state-of-the-art. Further, in some cases, it is

[3]We compare our results to commit 87cf8ec in [2]

TABLE I: New Best Area Results For The EPFL Suite

Benchmark	I/O	LUT-6 count	Level count
arbiter	256/129	**365**	117
div	128/128	**3267**	1211
i2c	147/142	**207**	15
log2	32/32	**6567**	119
max	512/130	**522**	189
mem_ctrl	1204/1231	**2086**	23
mult	128/128	**4920**	93
priority	128/8	**103**	26
sin	24/25	**1227**	55
hypotenuse	256/128	**40377**	4530
sqrt	128/64	**3075**	1106
square	64/128	**3242**	76

TABLE II: Smallest AIG Results For The EPFL Suite

Benchmark	I/O	Size AIG	Level count
arbiter	256/129	879	228
cavlc	10/11	483	78
div	128/128	19250	6228
i2c	147/142	710	25
log2	32/32	30522	348
mem_ctrl	1204/1231	7644	40
mult	128/128	25371	317
router	60/30	96	21
sin	24/25	4987	153
hypotenuse	256/128	209460	24926
sqrt	128/64	19706	5399
square	64/128	17010	343
voter	1001/1	9817	66

much smaller than the AIG size leading to the best LUT-6 results. As an example, we show 1.5× size reduction in the smallest known[4] AIG for the EPFL arbiter benchmark.

C. ASIC Results

We tested a commercial EDA flow, enhanced with the SBM framework, on 33 state-of-the-art ASICs, coming from major electronics industries. Due to non-disclosure agreements, we cannot provide details on each ASIC benchmark. However, we present average results w.r.t. a baseline flow without our Boolean methods. The post place & route results are summarized in Table III. All benchmarks are verified with an industrial formal equivalence checking flow.

Our complete design flow, embedding the new SBM framework, enables sensible combinational area & dynamic power (without considering the clock network) reductions, -2.20% and -1.15% respectively, on average, and also good WNS/TNS improvements, at only $+1.75\%$ runtime cost.

VI. Conclusions

In this paper, we presented the *Scalable Boolean Method* (SBM) framework, which consists of effective Boolean methods designed to be scalable in a modern synthesis flow. Within SBM, we presented (i) a generalized resubstitution framework based on computing and implementing the Boolean difference between two nodes, (ii) a gradient-based AIG optimization, (iii) a heterogeneous elimination for kerneling and

[4]The smallest known AIG for *arbiter* has been computed by strashing the previous best LUT6 result and running *resyn2rs* until no improvement is seen.

TABLE III: Post Place&Route Results on 33 Industrial Design

Flow	Comb. Area*	No-clk Dyn. Pow.*	WNS*	TNS*	Runtime
Baseline	1	1	1	1	1
Proposed flow	**-2.20%**	**-1.15%**	**-0.56%**	**-5.99%**	+1.75%

*"Comb. Area" is the combinational area, "No-clk Dyn. Pow" is the dynamic power of the circuit without considering the clock, "WNS" is the worst negative slack, and "TNS" is the total negative slack.

(iv) a revisiting of MSPF computation with BDDs. We showed significant synthesis results. We obtained strong improvements of the smallest known AIGs for EPFL benchmarks, and we improved 12 of the best known area results in the EPFL synthesis competition. We demonstrated -2.20% combinational area savings and -5.99% total negative slack reduction, after physical implementation, at contained runtime cost for a commercial EDA flow.

Acknowledgments

This research was supported in part by the Swiss National Science Foundation (200021-169084 MAJesty), by H2020-ERC-2014-ADG 669354 CyberCare, by the Defense Advanced Research Projects Agency (DARPA - FA8650-18-2-7849) and by SRC contracts 2710.001 and 2867.001.

References

[1] L. Amarú and et al., "Improvements to Boolean resynthesis," *Design, Automation and Test in Europe*, 2018.

[2] "https://github.com/lsils/benchmarks."

[3] R. K. Brayton and et al., "Multilevel logic synthesis," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 264–300, 1990.

[4] S. Muroga and et al., "The transduction method-design of logic networks based on permissible functions," *IEEE Trans. on Computers*, vol. 38, no. 10, pp. 1404–1424, 1989.

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

[6] R. K. Brayton and et al., "An exact minimizer for Boolean relations," *Int'l Conf. on Computer-Aided Design*, pp. 316–319, 1989.

[7] C. Yang and et al., "BDS: a BDD-based logic optimization system," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 21, no. 7, pp. 866–876, 2002.

[8] A. Mishchenko and et al., "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.

[9] K. Debnath and et al., "SAT-based redundancy removal," *Design, Automation and Test in Europe*, pp. 315–318, 2018.

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

[11] S. P. Khatri and et al., Eds., *Advanced Techniques in Logic Synthesis, Optimizations and Applications*. Springer, 2011.

[12] A. Mishchenko and et al., "DAG-aware AIG rewriting a fresh look at combinational logic synthesis," in *Design Automation Conference*, 2006, pp. 532–535.

[13] "ABC synthesis tool: https://github.com/berkeley-abc/abc."

[14] R. Drechsler and et al., *Binary decision diagrams: theory and implementation*. Springer Science & Business Media, 2013.

[15] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *Design Automation Conference*, 1990.

[16] L. Machado and et al., "Support-reducing functional decomposition for FPGA technology mapping," *Int'l Workshop on Logic and Synthesis*, 2018.