On-the-fly and DAG-aware: Rewriting Boolean Networks with Exact Synthesis

Heinz Riener[1] Winston Haaswijk[1] Alan Mishchenko[2] Giovanni De Micheli[1] Mathias Soeken[1]
[1]EPFL, Lausanne, Switzerland [2]UC Berkeley, CA, USA

***Abstract*—The paper presents a generalization of DAG-aware AIG rewriting for k-feasible Boolean networks, whose nodes are k-input lookup tables (k-LUTs). We introduce a high-effort DAG-aware rewriting algorithm, called *cut rewriting*, which uses exact synthesis to compute replacements on the fly, with support for Boolean don't cares. Cut rewriting pre-computes a large number of possible replacement candidates, but instead of eagerly rewriting the Boolean network, stores the replacements in a conflict graph. Heuristic optimization is used to derive a best, maximal subset of replacements that can be simultaneously applied to the Boolean network from the conflict graph. We optimize LUT mapped Boolean networks obtained from the ISCAS and EPFL combinational benchmark suites. For 3-LUT networks, experiments show that we achieve an average size improvement of 5.58% and up to 40.19% *after* state-of-the-art Boolean rewriting techniques were applied until saturation. Similarly, for 4-LUT networks, we obtain an average improvement of 4.04% and up to 12.60%.**

I. Introduction

Logic optimization of multi-level Boolean networks plays an important role in automated design flows for digital systems and is responsible for substantial area and delay reductions [1], [2]. These logic optimizations are typically carried out on a simple and technology-independent representation of the digital logic. Particularly, homogeneous data-structures, such as *and-inverter graphs* (AIGs) [3], [4]—being composed of two-input ANDs and inverters—or *majority-inverter graphs* (MIGs) [5]—being composed of majority-of-three gates and inverters—have been proven to be successful. Structural hashing on the intermediate representation ensures that no two nodes have identical incoming edges. Arbitrary Boolean networks can be transformed into AIGs or MIGs, for which a repertoire of scalable optimization techniques is available [6].

Boolean rewriting is an optimization technique that iteratively selects small parts of the Boolean network and replaces them with more compact implementations in order to reduce the overall number of nodes, while maintaining the global output functions of the Boolean network. An efficient implementation of this idea is DAG-aware AIG rewriting [7], which exploits structural hashing to find beneficial replacements that utilize the existing logic within the network. Being DAG-aware allows one to obtain a gain even when replacing a smaller part of logic by a larger one, by reusing already existing logic in the network. An efficient implementation of cut enumeration [8], [9], [10] in combination with fast truth table computations and a database of pre-computed replacements for a large number of Boolean functions makes the technique scalable.

In this paper, we generalize DAG-aware rewriting and present a DAG-aware rewriting algorithm that is directly applied to k-feasible Boolean networks (instead of AIGs). Replacements are computed on the fly using exact synthesis. Exact synthesis offers a more flexible, general, and scalable solution compared to a pre-computed database, and recent achievements in SAT-based exact synthesis enable its integration as an efficient engine in various logic synthesis applications [11]. As a consequence, the proposed approach is generic and capable of optimizing all common technology-independent logic representation including AIGs, MIGs, and XOR-based representations, as well as allows one to obtain size optimizations after technology mapping, e.g., in LUT mapping for FPGAs. Moreover, on the fly synthesis allows us to support don't care conditions, for which pre-computing a database is intractable. The approach is particularly useful as a post-optimization techniques when other resynthesis techniques saturate.

We call the new algorithm *cut rewriting*. The algorithm operates in two phases. In the first phase, a large number of possible replacement candidates is computed, but instead of eagerly rewriting the Boolean network, they are stored in a conflict graph. A node of the conflict graph denotes a possible replacement labeled with its achieved node reduction. An edge between two nodes denotes a conflict between two replacements such that only one of them can be applied. In the second phase, the conflict graph is used to determine a globally optimal subset of replacements by solving a maximum weighted vertex independent set problem. Note that, while we use exact synthesis to compute optimum replacement networks, the global optimization flow is heuristic.

We have implemented cut rewriting in a generic C++-17 open source Boolean network library and applied it to the ISCAS and EPFL combinational benchmarks. Experiments show that we achieve a reduction of 5.58% on average and up to 40.19% when resynthesizing 3-LUT networks. This is after state-of-the-art Boolean rewriting techniques, using the most effective optimization scripts for k-feasible Boolean networks in ABC [12], were applied until saturation. Similarly, for 4-LUT networks we achieve an average reduction of 4.04% and up to 12.60%

II. Preliminaries

A. Boolean networks

A *Boolean network* N is a *directed acyclic graph* (DAG). Each node corresponds to a logic gate. Each directed edge (n, m) is a wire connecting node n with node m. The *fanin*, respectively *fanout*, of a node $n \in N$ are the incoming, respectively outgoing, edges of the node. A Boolean network is *k-feasible* if the fanin size of all nodes is bounded by k. A k-LUT network is the most general k-feasible network in

which each gate can implement an arbitrary Boolean function. The *primary inputs* (PIs) are the nodes of the Boolean network without fanin. The *primary outputs* (POs) are the nodes of the Boolean network without fanout. All other nodes in the Boolean network are *gates*.

B. Cuts

A *cut* of a Boolean network is a pair (n, L), where n is a node, called *root*, and L is a set of nodes, called *leaves*, such that 1) each path from any PI to n passes through at least one leaf and 2) for each leaf $l \in L$, there is at least one path from a PI to n passing through l and not through any other leaf. The *cover* $\mathrm{Cover}(n, L)$ of a cut (n, L) is the set of all nodes $n \in N$ that appear on a path from any $l \in L$ to n including n, but excluding the leaves.

A fan-out free cone (FFC) of a node n is a cut $c = (n, L)$ such that no node $n' \in \mathrm{Cover}(n, L)$ with $n' \neq n$ has a parent node that is outside of $\mathrm{Cover}(n, L)$. The maximum fanout-free cone (MFFC, [8]) of a node n is its largest FFC. Informally, the MFFC of a node contains all the logic used exclusively by the node. When a node is removed or substituted, the logic in its MFFC can be removed [10].

C. Exact synthesis

Exact synthesis is the problem of finding the optimum Boolean network given a specification, where network optimality is defined with respect to some cost function. For example, we may want to find the network with the smallest number of nodes when we are optimizing for area, or the network with the fewest logic levels when optimizing for depth.

In recent years, there has been a substantial research effort into solving this problem using SAT based methods [13], [14], [15], [11]. Given an m-tuple of functions $f_1(x_1, \dots, x_n), \dots, f_m(x_1, \dots, x_n)$ over n variables, we can encode the following question as a SAT formula $\mathcal{F}_r$ [16], [17]:

> Does there exist a Boolean network N which implements $f_1, \dots, f_m$ using r gates?

If $\mathcal{F}_r$ is SAT, then such a network exists. The satisfying assignment corresponds to a network that implements the functions using r gates. Conversely, if $\mathcal{F}_r$ is UNSAT, then we have proven that no such network exists. Hence, if we initialize r to 0 and increment it until we find a satisfiable $\mathcal{F}_r$, we can use a SAT solver to find a provably size-optimum Boolean network for $f_1, \dots, f_m$. This process of size-optimum exact synthesis is illustrated in Fig. 1. More generally, we can re-formulate the above question with arbitrary cost functions $\mathcal{C}$.[1] We can then use a SAT solver to answer the question:

> Does there exist a Boolean network N which implements $f_1, \dots, f_m$ such that $\mathcal{C}(N) = r$?

There exist different ways of encoding the exact synthesis problem as SAT formulae, each having their own trade-offs. Some have fewer variables than others, but they may do so at the cost of adding more clauses. Unfortunately, there exists no comprehensive comparison of the differences in runtime between different encodings. We refer the interested reader to [16], [17], [13], [14], [15], [11] for detailed descriptions.

[1] Assuming the cost functions can be encoded as CNF formulae

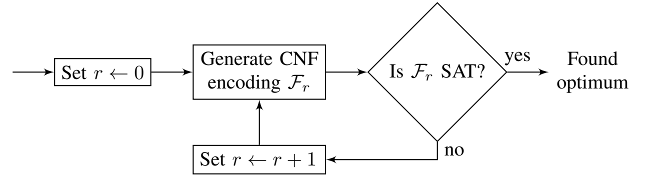


Fig. 1. Size-optimum SAT based exact synthesis.

Input: Boolean network N, cut size l, cut limit p
Output: Sorted list $C(n) = \{L_1, \dots, L_{p_n}\}$ of leaves for every node n in N
foreach *input n in N* **do** Set $C(n) \leftarrow \{\{n\}\}$;
foreach *gate n in N in topological order* **do**
    Let $n_1, n_2, \dots, n_m$ be the fanin nodes of n;
    foreach $L_1 \in C(n_1), L_2 \in C(n_2), \dots, L_m \in C(n_m)$ **do**
        Set $L = L_1 \cup L_2 \cup \dots \cup L_m$;
        if $|L| \geq l$ **then continue**;
        if $\exists L' \in C(n) : L' \subseteq L$ **then continue**;
        Remove all L' from $C(n)$ for which $L \subset L'$;
        Insert L into $C(n)$ and keep $C(n)$ sorted;
    if $|C(n)| > p$ **then**
        Remove the last $|C(n)| - p$ elements from $C(n)$;

Algorithm 1: Cut enumeration

III. Boolean Rewriting

In this section, we introduce a Boolean rewriting algorithm called *cut rewriting*, which is directly applied to k-feasible Boolean networks. Cut rewriting works in two phases. In the first phase, potential network replacements for a large number of subgraphs are pre-computed using exact synthesis. Replacements that reduce the overall network size are stored in a conflict graph. A node of the conflict graph denotes a possible rewriting labeled with its achieved gain in node reduction. An edge between two nodes denotes a conflict between two replacements such that only one of them can be applied. In the second phase, a maximal set of non-conflicting replacements are heuristically selected from the conflict graph to maximize the overall gain and applied to rewrite the Boolean network.

A. Cut enumeration

The proposed rewriting algorithm makes use of cut enumeration, which is an algorithm that can compute all cuts of all nodes in a Boolean network. Since the number of cuts is very large, the number of enumerated cuts is bounded by a parameter l for the cut size and a parameter p for the maximum number of cuts for each node. This technique is referred to as priority cuts [10] as it selects the subset of all cuts with respect to some cost function, in our case the number of the cuts' leaves. The returned cut sets are also irredundant and do not contain two cuts (n, L_1) and (n, L_2) such that L_1 dominates L_2, i.e., $L_1 \subseteq L_2$.

Algorithm 1 sketches the cut enumeration procedure that is used in the rewriting algorithm. It omits details on truth table computation and cut pruning based on functional dependence. The algorithm returns on termination a map from node n to a sorted list of leaves $C(n)$ such that every pair (n, L) for $L \in C(n)$ is a cut of the Boolean network. In addition to basic

function DerefNode(n, L)
 if $n \in L$ **then return** 0;
 Set value $\leftarrow 1$;
 foreach *child* c *of* n **do**
  Set ref(c) $\leftarrow$ ref(c) $- 1$;
  if ref(c) $= 0$ **then**
   Set value $\leftarrow$ value $+$ DerefNode(c, L);
 return value;

Algorithm 2: Dereferencing a node

function RefNode(n, L)
 if $n \in L$ **then return** 0;
 Set value $\leftarrow 1$;
 foreach *child* c *of* n **do**
  Set ref(c) $\leftarrow$ ref(c) $+ 1$;
  if ref(c) $= 1$ **then**
   Set value $\leftarrow$ value $+$ RefNode(c, L);
 return value;

Algorithm 3: Referencing a node

cut function computation, we also compute the cut function's Boolean controllability don't cares, which are based on the local structure of the logic network.

B. DAG-aware rewriting

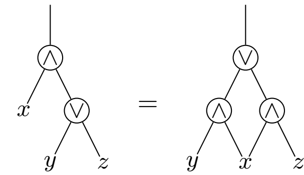
In this section, we review an efficient algorithm to compute the gain of replacing a part of logic in a network by another part of logic [7].

The algorithm to compute the gain makes use of reference counting and assigns a value to each node in the network. These values are initialized with the nodes' fanout sizes. New nodes that are added to the network for a possible replacement will be assigned a reference count of 0. The reference count of a node indicates how many other nodes require this node in the network. In particular, a reference count of 0 means that the node is not required in the network. The algorithm also exploits structural hashing, i.e., nodes from a replacement candidate that are already in the network will not be added another time, and also its reference counter will not be changed.

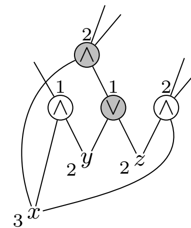
For "simulating" the removal of a node n from a network, we recursively decrement all predecessors in the transitive fanin of the node and continue as long as the reference counters of a child become 0 or a leaf node is reached. Algorithm 2 shows the details. It receives as inputs the node n and the leaves of a cut L.

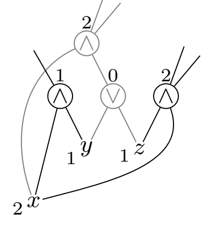
Adding a node to a network can be "simulated" by the inverse algorithm to DerefNode, called RefNode (see Algorithm 3), which will increment reference counters and continue on the predecessors as long as the reference counter was 0 before incrementing it, and stops otherwise or when it reaches a leave node.

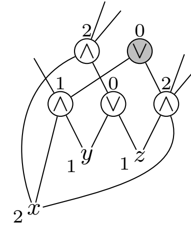
Fig. 2 shows the gain calculation of a replacement from one cut into another: Fig. 2(a) shows two functionally equivalent structures. The cut on the left is already contained in the network shown in Fig. 2(b). Fig. 2(b) also shows the initial reference counters which are equal to the fanout size of each node. Calling DerefNode on the top most AND gate changes the references counters as shown in Fig. 2(c). In particular,

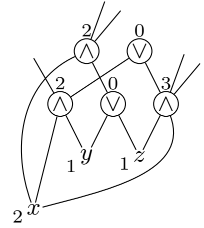


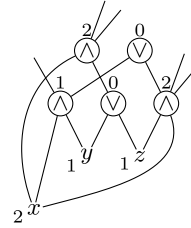
(a) Replacement candidate


(b) Initial network


(c) Deref original cut, obtained value 2


(d) Insert replacement candidate


(e) Ref replacement cut, obtained value is 1.


(f) Deref replacement cut, obtained value is 1.

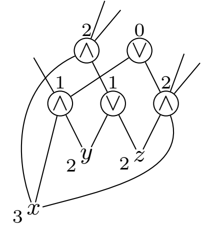

(g) Ref original cut, obtained value is 2.

Fig. 2. Example for estimating the insertion of a replacement cut using reference counters.

the OR gate in the middle of the network now has a reference value of 0, meaning it is not required anymore after deleting the cut. Together with the root node this leads to a value of 2 which is returned by DerefNode. Afterwards the logic for the replacement cut is added in Fig. 2(d). Note that two of the three gates are already present in the network and only one new node is added, which is initialized with a reference value of 0. All other reference values remain the same. Calling RefNode on the root node of the inserted cut simulates an insertion of the cut and leads to the reference values as in Fig. 2(e). The function returns 1 for the increment of the root node. From these two values we can derive that replacing the first cut by the other will save $2 - 1 = 1$ nodes. Since the cost of the replacement should only be calculated and not actually be performed one can undo the changes to the reference counters by simply calling the inverse functions in inverse order, i.e., calling DerefNode on the root node of the replacement cut and RefNode on the root node of the original cut leading to the reference values as shown in Fig. 2(f) and Fig. 2(g), respectively.

```
function DryReplace(N, n ↦ n′, L)
    Set v1 ← DerefNode(n, L);
    Insert cut (n′, L) into the network;
    Set v2 ← RefNode(n′, L);
    DerefNode(n′, L);
    RefNode(n, L);
    return v1 − v2;
```

Algorithm 4: Adding a new cut (n', L) into the network and calculating the gain when replacing an existing cut (n, L)

```
function MFFCSize(N, n)
    Set L ← primary inputs of N;
    Set v ← DerefNode(n, L);
    RefNode(n, L);
    return v;
```

Algorithm 5: Compute the size of the MFFC of n

This example motivates a function called $\mathrm{DryReplace}(N, n \mapsto n', L)$, as shown in Algorithm 4, that inside a network N simulates the replacement of an existing cut (n, L) with a new cut (n', L) by using reference counters. The algorithm does not change the reference values of existing nodes in N and all newly added nodes will be assigned a reference value of 0. The function returns the gain of replacing the existing cut with the new one. This gain may be negative.

The routines RefNode and DerefNode can also be used conveniently to compute the size of the MFFC of a node, as shown in Algorithm 5. In here, the cut leaves L are the primary inputs in order to find all logic in the node's MFFC.

C. Cut rewriting

This section describes a rewriting algorithm that finds replacement candidates for all enumerated cuts in a k-feasible Boolean network. Since cuts found by cut enumeration may not be an FFC, DAG-aware rewriting techniques are used to compute the gain of possible replacement candidates. After all replacement candidates and their gain have been computed, the algorithm finds a set of replacement candidates that maximize the overall gain. Algorithm 6 shows a pseudo code for the algorithm, which is explained in detail in the remainder of this section.

The algorithm starts by computing all cuts for a cut size l and cut limit p. The cut size should be chosen according to k. For example, l must be larger than k to find replacement candidates that lead to a gain, but if l is too large it can significantly degrade the success rate of exact synthesis. We experimentally evaluated that for $k = 2$ and $k = 3$, cut sizes $l = 5$ and $l = 6$ lead to good results, respectively.

Next, an empty graph G is initialized that will be constructed when enumerating replacement candidates for the cuts. The graph has vertices V for cuts, and an edge in E if two cuts have overlapping logic and can therefore not be replaced simultaneously. Also it has a vertex weight w that is assigned the possible gain of a cut when being replaced by its best found replacement. Finally, the mapping r maps a vertex to the root node of the best replacement cut.

```
Input: Boolean network N, cut size l, cut limit p
Set C ← CutEnumeration(N, l, p);
Set G ← (V = ∅, E = ∅, w, r);
foreach gate n ∈ N do
    if MFFCSize(N, n) = 1 then continue;
    foreach leaves L ∈ C(n) do
        Set bestGain ← 0;
        Set bestReplacement ← Λ;
        foreach replacement (n′, L) do
            Set gain ← DryReplace(N, n ↦ n′, L);
            if gain > bestGain then
                Set bestGain ← gain;
                Set bestReplacement ← n′;
        if bestReplacement ≠ Λ then
            Add vertex v = (n, L) to V;
            Set w(v) ← bestGain;
            Set r(v) ← bestReplacement;
foreach L1 ∈ C(n1) and L2 ∈ C(n2) do
    if Cover(n1, L1) ∩ Cover(n2, L2) ≠ ∅ then
        Add edge (n1, L1) — (n2, L2) to E;
Set V′ ← MaximalIndependentVertexSet(G);
foreach (n, L) ∈ V′ do
    Replace(N, n ↦ r(n), L);
```

Algorithm 6: Cut rewriting

For each cut (n, L) the algorithm enumerates possible replacements (n', L) using SAT-based exact synthesis. The replacements must not necessarily be optimum in size. The runtime of exact synthesis can be controlled by setting thresholds on the conflict limit of the SAT solver [11]. For each replacement candidate the gain is computed using DryReplace and the best gain is stored in a variable gain together with the best replacement candidate in bestReplacement. If a replacement that leads to a gain can be found a vertex for the cut is added to G and the mappings w and r are updated with the gain and the replacement candidate, respectively. Afterwards, edges are added to G for each two cuts that have overlapping covers. To obtain a good subset of non-conflicting replacement candidates we heuristically solve the maximum weighted independent vertex set problem on G with respect to weights w using the greedy algorithm GWMIN [18], which provides an approximation guarantee of finding a solution with a weight of at least $\frac{1}{\Delta}\alpha(G)$, where Δ is the degree of G and $\alpha(G)$ is the weight of the exact solution.

In order to speed up computation, we apply two effective techniques. First, we skip all nodes whose MFFC size is 1, i.e., the replacement of the node cannot lead to any positive gain. Second, we cache all replacement candidates computed by exact synthesis. That is, for every unique cut function, replacements are only computed once and then stored in a hash table. This table is particularly useful, when calling cut rewriting repeatedly, because successive runs need to call exact synthesis only on new cuts found by cut enumeration.

IV. Experiments

We implemented our approach in C++-17 using the EPFL logic synthesis libraries [19] *mockturtle*[2] and *percy*[3] in a

[2]see github.com/lsils/mockturtle

[3]see github.com/whaaswijk/percy

generic way such that it can in principle be applied to any k-LUT network. Exact synthesis has the most impact to performance, and our experiments indicate that using the current implementation, practical and scalable results can be achieved.

We applied cut rewriting to improve the size of 3-LUT and 4-LUT networks for the combinational instances in the ISCAS benchmarks and the arithmetic instances in the EPFL benchmarks [20]. The baseline networks are obtained by performing a LUT mapping using '`&if -K k`' with $k \in \{3, 4\}$ in ABC [12], respectively. In case of the EPFL benchmarks, we chose the best-known size-optimized 6-LUT benchmarks as a starting point.[4] As state-of-the-art area optimization we apply a synthesis script that interleaves priority-cut-based LUT mapping ('`&if`') [10], structural choices ('`&dch`' and '`&synch2`') [21], [22], and Boolean network optimization and resynthesis ('`&mfs`') [23]. We apply the synthesis script

```
&st; &synch2; &if -m -a -K k; &mfs -W 10;
  &st; &dch; &if -m -a -K k; &mfs -W 10
```

with the respective k parameter ten times and pick the best result that was encountered during all iterations. This optimization method is called *MFS* in the remainder.

We call the proposed cut rewriting algorithm repeatedly until no further gain in area can be achieved. We apply our approach as a post-optimization approach on the optimized networks obtained by MFS.

Tables I and II show the experimental results for 3-LUTs and 4-LUTs, respectively. The table lists the baseline, the results obtained after MFS, and the results obtained by applying cut rewriting after MFS. For each it lists the number of gates and the number of logic levels. It also lists runtime in seconds. In case of *MFS + Cut rewriting* it only lists the time required by cut rewriting. The last column shows the improvement that can be obtained by calling cut rewriting on the results already optimized by MFS. The cut size and cut limit for cut enumeration are $l = 6$ and $p = 12$, respectively. We compute one replacement candidate for each cut using exact synthesis with a conflict limit of 1000.

The strength of cut resynthesis becomes evident when used as a post-optimization method after MFS has been tried heavily to find the best network. On top of the significantly improved results, cut rewriting can find additional improvement—often even with a comparably small runtime overhead. The average improvement is 5.58% and 4.04% when resynthesizing 3-LUT and 4-LUT networks, respectively. The best improvement is achieved for the 128-bit adder, which improved by 40.19% when considering 3-LUT networks. Starting from a baseline implementation that has 67 logic levels it manages to regain the size-optimal carry ripple implementation with one sum gate (XOR-3) and one carry gate (majority-3) for each pair of input bits.

V. Conclusion

We have presented a DAG-aware rewriting algorithm directly applied to k-feasible Boolean networks. Instead of using simple transformations, a SAT-based exact synthesis engine with support for Boolean don't cares is employed for rewriting existing logic. Moreover, instead of applying replacements ad-hoc, all possible gains are stored in a conflict graph, from which a maximal subset of compatible replacements is computed. We show size improvements up to 40.19% and 12.60% when resynthesizing 3-LUT networks and 4-LUT networks, respectively. Today's exact synthesis methods are not strong enough to efficiently find compact 6-LUT networks for functions with 8–10 variables. Improvements to the scalability of exact synthesis can be directly exploited by our proposed approach.

Acknowledgments: This research was supported by the Swiss National Science Foundation (200021-169084 MAJesty), by the European Research Council in the project H2020-ERC-2014-ADG 669354 CyberCare and by SRC contracts "SAT-based methods for scalable synthesis and verification" and "Deep integration of computation engines for scalability in synthesis and verification."

[4] see github.com/lsils/benchmarks

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] L. Hellerman, "A catalog of three-variable Or-invert and And-invert logical circuits," *IEEE Trans. Electronic Computers*, vol. 12, no. 3, pp. 198–223, 1963.

[4] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust boolean reasoning for equivalence checking and functional property verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1377–1394, 2002.

[5] L. Amarú, P. Gaillardon, and G. De Micheli, "Majority-Inverter Graph: A New Paradigm for Logic Optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 5, pp. 806–819, 2016.

[6] 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.

[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] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," in *Int'l Symp. on Field Programmable Gate Arrays*, 1999, pp. 29–35.

[9] J. Cong and Y. Ding, "On area/depth trade-off in LUT-based FPGA technology mapping," *IEEE Trans. VLSI Syst.*, vol. 2, no. 2, pp. 137–148, 1994.

[10] A. Mishchenko, S. Cho, S. Chatterjee, and R. K. Brayton, "Combinational and sequential mapping with priority cuts," in *Int'l Conf. on Computer-Aided Design*, 2007, pp. 354–361.

[11] W. Haaswijk, A. Mishchenko, M. Soeken, and G. De Micheli, "SAT Based Exact Synthesis Using DAG Topology Families," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18, 2018.

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

[13] 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, pp. 151–156.

[14] 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*, vol. 36, no. 11, pp. 1842–1855, 2017.

[15] M. Soeken, G. De Micheli, and A. Mishchenko, "Busy man's synthesis: Combinational delay optimization with SAT," in *Design, Automation and Test in Europe*, 2017, pp. 830–835.

[16] N. Éen, "Practical SAT - a tutorial on applied satisfiability solving," 2007, slides of invited talk at FMCAD.

TABLE I
EXPERIMENTAL RESULTS FOR 3-LUT RESYNTHESIS

Name	Baseline				MFS			MFS + Cut rewriting			Improvement
	PIs	POs	gates	levels	gates	levels	time	gates	levels	time	
c432	36	7	113	16	71	19	0.66	**68**	22	1.17	4.23%
c499	41	32	112	9	102	9	2.27	102	9	0.51	0.00%
c880	60	26	175	13	141	14	1.87	**139**	14	2.65	1.42%
c1355	41	32	112	9	102	9	1.79	102	9	0.55	0.00%
c2670	157	64	304	11	216	12	2.02	**211**	14	2.32	2.31%
c3540	50	22	563	19	316	20	5.33	**309**	20	9.16	2.22%
c5315	178	123	838	14	521	15	5.89	**510**	15	12.01	2.11%
c6288	32	32	733	31	748	33	30.81	748	33	0.02	0.00%
c7552	207	108	666	14	540	32	6.12	**522**	34	17.32	3.33%
adder	256	129	827	67	428	85	4.92	**256**	128	1.08	40.19%
bar	135	128	1018	7	896	7	10.13	896	7	0.00	0.00%
div	128	128	13202	2299	8465	2170	136.41	**7010**	2290	4030.98	17.19%
log2	32	32	21759	216	15927	201	588.34	**15146**	195	22650.17	4.90%
max	512	130	891	249	823	249	9.56	**808**	251	5.25	1.82%
multiplier	128	128	18983	147	11346	141	366.53	**11196**	145	9771.94	1.23%
sin	24	25	4334	99	2989	102	907.73	**2851**	99	308.27	4.62%
sqrt	128	64	12918	2116	8031	2147	124.13	**7010**	2174	3115.73	12.71%
square	64	128	15290	168	6931	165	446.19	**6789**	166	188.10	2.05%
Average											5.58%
Sum							2650.00			40117.26	

TABLE II
EXPERIMENTAL RESULTS FOR 4-LUT RESYNTHESIS

Name	Baseline				MFS			MFS + Cut rewriting			Improvement
	PIs	POs	gates	levels	gates	levels	time	gates	levels	time	
c432	36	7	100	10	52	16	1.52	52	16	0.19	0.00%
c499	41	32	78	5	78	6	5.17	78	6	0.02	0.00%
c880	60	26	125	9	108	14	4.14	**106**	14	0.29	1.86%
c1355	41	32	78	5	80	6	5.74	80	6	0.21	0.00%
c2670	157	64	204	7	178	9	7.98	**161**	9	0.26	9.55%
c3540	50	22	348	12	236	16	14.70	**231**	16	1.69	2.12%
c5315	178	123	506	10	425	12	17.28	**383**	12	1.14	9.88%
c6288	32	32	503	25	494	31	89.78	494	31	0.10	0.00%
c7552	207	108	520	8	427	24	17.40	**424**	24	1.11	0.70%
adder	256	129	529	44	341	84	15.96	**298**	127	0.06	12.60%
bar	135	128	1018	7	896	7	35.98	896	7	0.00	0.00%
div	128	128	9597	1486	5113	2007	390.33	**4681**	2069	27.28	8.45%
log2	32	32	14021	128	11659	172	1993.53	**10761**	166	14097.98	7.70%
max	512	130	1074	135	785	245	27.62	**784**	245	1.02	0.13%
multiplier	128	128	11256	98	8264	138	1223.32	**8084**	137	1893.17	2.61%
sin	24	25	2921	62	2172	88	4393.63	**2056**	87	60.49	5.34%
sqrt	128	64	9139	1386	5004	1945	384.45	**4534**	1992	30.46	9.21%
square	64	128	8843	104	5737	132	1153.30	**5588**	140	35.44	2.60%
Average											4.04%
Sum							9781.84			16150.65	

[17] A. Kojevnikov, A. S. Kulikov, and G. Yaroslavtsev, “Finding efficient circuits using SAT-solvers,” in *Int’l Conf. on Theory and Applications of Satisfiability Testing*, 2009, pp. 32–44.

[18] S. Sakai, M. Togasaki, and K. Yamazaki, “A note on greedy algorithms for the maximum weighted independent set problem,” *Discrete Applied Mathematics*, vol. 126, no. 2–3, pp. 313–322, 2003.

[19] M. Soeken, H. Riener, W. Haaswijk, and G. De Micheli, “The EPFL logic synthesis libraries,” in *Int’l Workshop on Logic and Synthesis*, 2018, arXiv preprint arXiv:1805.05121.

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

[21] S. Chatterjee, A. Mishchenko, R. K. Brayton, X. Wang, and T. Kam, “Reducing structural bias in technology mapping,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 25, no. 12, pp. 2894–2903, 2006.

[22] A. Mishchenko, S. Chatterjee, and R. K. Brayton, “Improvements to technology mapping for LUT-based FPGAs,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 240–253, 2007.

[23] 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.