Exact DAG-Aware Rewriting

Heinz Riener[1] Alan Mishchenko[2] Mathias Soeken[3]
[1]Integrated Systems Laboratory, EPFL, Lausanne, Switzerland
[2]Electrical Engineering and Computer Sciences, University of California, Berkeley, USA
[3]Microsoft, Switzerland

***Abstract*—We present a generic resynthesis framework for optimizing Boolean networks parameterized with a multi-level logic representation, a cut-computation algorithm, and a resynthesis algorithm. The framework allows us to realize powerful optimization algorithms in a plug-and-play fashion. We show the framework's versatility by composing an exact DAG-aware rewriting engine. Disjoint-support decomposition and SAT-based exact synthesis together with efficient caching strategies enable the algorithm to resynthesize larger parts of the logic. DAG-aware rewriting is used to compute the gain of resynthesis while taking the benefit of structural hashing into account.**

I. Introduction

Logic synthesis and logic optimization techniques play a key role in the automated design of all digital systems and are often capable of substantially reducing area and delay requirements of a circuit under design. In particular, resynthesis methods based on Boolean reasoning on homogeneous multi-level logic representations [1] have proven effective, and are today indispensable in any complex design [2] or engineering change order (ECO) flow [3], [4], [5].

Under the hood, state-of-the-art resynthesis frameworks [6], [7] rely on an interplay of scalable structural analysis and peephole optimization algorithms. Such algorithms first partition the given Boolean network into small subnetworks, identify the output functions implemented by the subnetworks, and replace them with size-optimum solutions obtained from a pre-enumerated database. Effective strategies for pre-enumerating and storing databases of optimum circuitry for all Boolean functions up to 5-input variables exist [8]. Resynthesizing larger subnetworks requires on-the-fly computation [9] of replacement candidates using, e.g., SAT-based exact synthesis methods [10], which consequently challenges the efficacy and scalability of resynthesis.

The rise of cloud-based software tools in IT, and in particular in the EDA industry [11], promises significant more computational power and memory resources readily available for future algorithms. The memory resources of realistic cloud-based EDA solutions have been estimated in [11] to amount up to 20 petabytes. This estimation is based on a comparison to the traffic analysis services offered by Google maps. The game-changing opportunity offered by cloud services allows us to design a new generation of logic synthesis algorithms that can make heavily use of large-scale caching and multi-core parallelization. We envision a logic resynthesis framework capable of resynthesizing larger subnetworks using exact synthesis—each computed solution (as well as a timeout) is cached, such that all exact synthesis queries have to be solved only once. This strategy is supported by the fact that companies often work on the same design for many months and run logic synthesis algorithms on a daily or weekly basis. A first run of logic synthesis may be slow. However, with every rerun of the synthesis engine, the cache will be populated with more solutions such that the runtime over the project time quickly amortizes.

In this paper, we present a generic resynthesis framework for optimizing Boolean networks parameterized with a multi-level logic representation, a cut-computation algorithm, and a resynthesis algorithm. This framework allows us to realize powerful optimization algorithms in a plug-and-play fashion. Different resynthesis algorithms are available including algorithms based on recursive top-down disjoint-support decomposition, exact synthesis, and database lookups of best-known realizations. The implementation is available online as a part of the EPFL logic synthesis libraries [12].

II. Background

A *Boolean network* (or *circuit*) N is a *directed acyclic graph* (DAG) with nodes corresponding to Boolean functions and edges corresponding to wires connecting them. The *fanin*, respectively, *fanout* of a node $n \in N$ are the incoming, respectively, outgoing edges of n. The *primary inputs* (PIs) are the nodes of the Boolean network without fanin. The *primary outputs* (POs) are a subset of the nodes connecting the Boolean network with its environment. A k-bounded *transitive fanin-cone* $\mathrm{TFI}_k(n)$ and k-bounded *transitive fanout-cone* $\mathrm{TFO}_k(n)$ of a node n are the subsets of the nodes reachable through traversing at most k transitive fanin-edges and at most k transitive fanout-edges of n. Without loss of generality, the transitive fanin-cone and transitive fanout-

cone, respectively, can be generalized to sets $N' \subset N$ of nodes in N, such that $\mathrm{TFI}_k(N') = \bigcup_{n \in N'} \mathrm{TFI}_k(n)$ and $\mathrm{TFO}_k(N') = \bigcup_{n \in N'} \mathrm{TFO}_k(n)$.

A *cut* $C = (r, L)$ of a Boolean network N is a pair, where r is a node, called *root*, and L is a set of nodes, called *leaves*, such that

1) each path from any PI in N to r passes through at least one leaf in L and
2) for each leaf $l \in L$, there is at least one path from a PI to r passing through l and not through any other leaf.

The *cover* $N.\mathrm{cover}(C)$ of a cut $C = (r, L)$ in N is the set of all nodes $n \in N$ that appear on a path from any $l \in L$ to r including r, but excluding the leaves. The cover $N.\mathrm{cover}(W)$ of a pair $W = (R, L)$, where R and L are two node sets called *roots* and *leaves*, respectively, is the union $\bigcup_{r \in R}(r, L)$ of all covers of cuts (r, L) for $r \in R$.

A *fanout-free cone* (FFC) of a node r is a cut (r, L) such that no node $n \in N.\mathrm{cover}(C)$ with $n \neq r$ has a fanout node that is outside of $N.\mathrm{cover}(C)$. The *maximum fanout-free cone* (MFFC, [13]) of a node r, denoted by $N.\mathrm{mffc}(r)$, is its largest FFC. The MFFC of a node r is unique and contains all the logic used exclusively by this node. If the node is substituted or removed from the network all nodes in its MFFC can be removed.

We use Boolean chains to formalize exact synthesis of Boolean networks. A k-input operator *Boolean chain* with p inputs $x = x_1, \ldots, x_p$ and q outputs $f = f_1(x), \ldots, f_q(x)$ is a sequence $x_{p+1}, \ldots, x_{p+s}$, where

$$x_i = \phi_i(x_{j(i,1)}, \ldots, x_{j(i,k)}) \text{ for } p + 1 \leq i \leq p + s$$

such that $\phi_i : \mathbb{B}^k \to \mathbb{B}$ is a k-input Boolean function, $1 \leq j(i, \cdot) < i$, and for all $1 \leq k \leq q$ either $f_k(x) = x_{l(k)}$ or $f_k(x) = \bar{x}_{l(k)}$, where $0 \leq l(k) \leq p + s$, and $x_0 = 0$ denotes the constant zero input. We call s the *length* of the Boolean chain and $f : \mathbb{B}^p \to \mathbb{B}^q$ the (multi-output) Boolean function *implemented* by the Boolean chain. A Boolean chain is *minimum-length* if no other Boolean chain exists that implements the same Boolean function f with smaller length. Minimum-length Boolean chains are not unique.

III. Boolean Resynthesis Framework

In this section, we describe a generic and flexible resynthesis framework for optimizing Boolean networks. The framework is parameterized with a multi-level logic representation, a cut-computation algorithm, and a resynthesis algorithm. The described approach is generic [7] and applicable to Boolean networks provided in form of a multi-level logic representation, such as an And-Inverter Graph (AIG) or a Majority-Inverter Graph (MIG), or a k-input lookup-table (LUT) network.

Algorithm 1 summarizes the overall process of resynthesis for a given Boolean network N. The network is optimized "inplace"—subnetworks are iteratively extracted by a cut computation algorithm, simulated to determine the Boolean functions at the subnetwork's outputs, and resynthesized. A subnetwork C is substituted with its resynthesized counterpart C' if C' is preferred over C with respect to a cost function. The usual cost functions consider the network's number of nodes (as a technology-independent indicator for area requirements) or the network's longest path from the PIs to the POs (as a technology-independent indicator of the delay requirements). An additional Boolean flag z provided as input to the algorithm allows us to enable zero-gain rewriting, which is often useful to restructure a Boolean network N and to step out of local minima during optimization.

Algorithm 1: Boolean resynthesis framework

```
Data: Boolean network N, Boolean flag z
Result: Optimized network N
foreach n ∈ N do
    C ← ExtractSubnetwork(n);
    f ← Simulate(C);
    C' ← Resynthesize(f);
    g ← Gain(N, C, C');
    if (g > 0) then
        N.substitute(C, C');
    else if (z ∧ g = 0) then
        N.substitute(C, C');
return N;
```

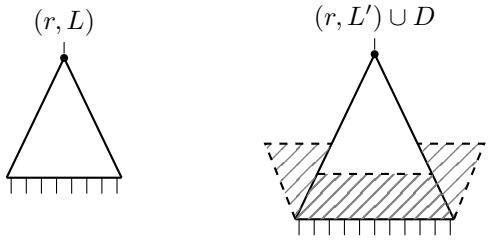

Fig. 1. Extended fanout-free cut to extract additional divisors.

A. Cut computation

Extracting subnetworks from a Boolean network is an important step in many resynthesis frameworks. A simple, yet powerful, way to extract a fraction of nodes is commonly known as *cut computation*. Cut-computation algorithms exist in many variations. Given a single node $n \in N$ in a Boolean network N, a cut-computation algorithm computes one (or more) sets $L_1, \ldots, L_l$ of leaf nodes such that each pair (n, L_i) for $1 \leq i \leq l$ is a cut. Many cut-computation algorithms take additional parameters to constrain the shape of the computed cuts. Depending on the application, for instance, one may desire to limit the number of leaves in a cut, the number of nodes in the cover of a cut, or the number of levels in a cut.

We present a simple algorithm to compute cuts and a strategy for growing them to extract additional divisors from their surroundings depicted in Fig. 1.

Fanin cuts. Algorithm 2 shows an eager strategy to expand a given leaf set in fanin-direction. Starting with the singleton

Algorithm 2: ExpandLeaves

Data: Boolean network N, set L of nodes, cut size k
Result: Expanded leaf set L
foreach $l \in L \cup \{\bot\}$ **do**
  if $N.\mathrm{isPI}(l)$ **and** $|L| - 1 + |N.\mathrm{fanin}(l)| \leq k$ **then**
    break;
if $l = \bot$ **then return** L;
$L \leftarrow L \backslash \{l\}$
foreach $n \in N.\mathrm{fanin}(l)$ **do**
  $L \leftarrow L \cup \{n\}$;
return L;

Algorithm 3: ExtendedCut

Data: Boolean network N, set L of nodes, cut sizes k
Result: Expanded leaf set L', set D of additional divisors
$L' \leftarrow \mathrm{ExpandLeaves}(N, L, k)$;
$D \leftarrow N.\mathrm{cover}(L, L')$
foreach $n \in D$ **do**
  foreach $d \in N.\mathrm{fanout}(n)$ **do**
    if $d \in D \cup \mathrm{TFO}_\infty(L)$ **then continue**;
    if $N.\mathrm{fanin}(d) \subseteq D$ **then**
      $D \leftarrow D \cup \{d\}$
$D \leftarrow D \backslash \big(\bigcup_{l \in L} N.\mathrm{mffc}(l)\big)$;
return L', D;

$L = \{r\}$, $r \in N$, the algorithm iteratively expands L by substituting leaves in L by their respective fanins. The expansion process stops at a leaf if (i) the leaf is a primary input and has no fanin or (ii) the size of L would exceed the cut's size limitation k if further expanded at this leaf. As output, the algorithm computes a set L' of leaves such that (r, L') is a cut and $|L'| \leq k$.

The described expansion algorithm can start from a singleton with one root r to compute a cut with a single output r or from a set of roots to produce a multi-output cut.

Cut extension. In several applications, extracting additional divisors from the surroundings of a cut is useful without requiring the divisors being fanout-free themselves. Algorithm 3 shows a cut-computation strategy that greedily expands a node set L in the fanin-direction using Algorithm 2. Afterward, the algorithm iterative merges fanouts to the cover of the extended cut if (i) they are not part of the transitive-fanout cone of L and (ii) all their fanins are in the cover or already considered as additional divisors. The first condition guarantees that the DAG remains acyclic. The second condition ensures that the divisors depend only on the leaves of the extended cut.

B. Exact synthesis

Exact synthesis is the problem of finding an *optimum* implementation of a given specification. In the context of logic synthesis, the specification is a Boolean function to implement by a Boolean network with additional constraints such as upper and lower bounds on the number of gates or the number of levels or restrictions on the type of gates to use.

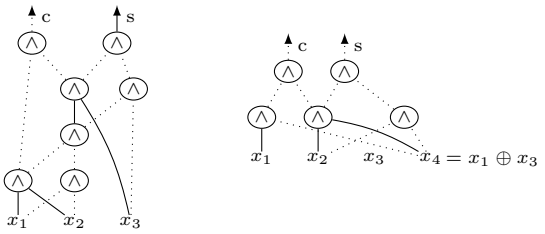

Fig. 2. Minimum-length Boolean chains implementing a full adder with the three inputs x_1, x_2, and x_3 (left) and the additional known function $x_4 = x_1 \oplus x_3$ (right).

There is substantial research work in solving such constrained logic synthesis problems using SAT-based methods. The constraints are encoded into a propositional logic formula over Boolean variables in such a way that every satisfying assignment of the formula corresponds to a Boolean network that meets the desired constraints. More formally, given a multi-output Boolean function $f : \mathbb{B}^p \to \mathbb{B}^q$ over inputs $x_1, \ldots, x_p$ as specification, a constraint satisfaction problem can be formulated that asks for the existence of a minimum-length Boolean chain $x_{p+1}, \ldots, x_{p+s}$ that implements f [10].

Exact synthesis with known functions. We consider the following variation of the exact synthesis problem for k-input operator Boolean chains: given a (multi-output) Boolean function $f : \mathbb{B}^p \to \mathbb{B}^q$ over inputs $x_1, \ldots, x_p$ as specification and additional single-output functions $x_{p+1}(x), \ldots, x_{p+r}(x)$ (depended only on x), find a minimum-length Boolean chain $x_{p+r+1}, \ldots, x_{p+r+s}$ with p inputs, r additional functions, and q outputs that implements f, where

$$x_i = \phi_i(x_{j(i,1)}, \ldots, x_{j(i,k)}) \text{ for } p + r + 1 \leq i \leq p + r + s$$

such that $\phi_i : \mathbb{B}^k \to \mathbb{B}$ is a k-input Boolean function, $1 \leq j(i, \cdot) < i$, and for all $1 \leq k \leq q$ either $f_k(x) = x_{l(k)}$ or $f_k(x) = \bar{x}_{l(k)}$, where $0 \leq l(k) \leq p + r + s$, and $x_0 = 0$ denotes the constant zero input.

The proposed formulation of the exact synthesis problem allows us to consider additional known functions during synthesis. As a simple example, suppose that a Boolean chain over the three inputs x_1, x_2, and x_3 for the specification of a full adder should be synthesized. The full adder has two outputs $c = \mathrm{maj}_3(x_1, x_2, x_3)$ (carry) and $s = x_1 \oplus x_2 \oplus x_3$ (sum), where maj_3 is the three-input majority operation, and the Boolean chain should only consist of And-gates and inverters. Fig. 2 (on the left) shows a minimum-length Boolean chain that implements this specification. Each node in the graph denotes an And-gate. Regular edges are shown with solid lines, dotted lines are used to denote inverters. Now, suppose that additionally the Boolean function $x_4 = x_1 \oplus x_3$ is available and can be used during synthesis, e.g., because a divisor with this Boolean function exists already in the Boolean network. Fig. 2 (on the right) shows the corresponding Boolean chain with 5 nodes, where the known function is represented as an additional input x_4.

Algorithm 4: DerefNode

Data: Boolean network N, node n, leaf set L
Result: Integer value
if $n \in L$ **then return** 0;
value $\leftarrow$ 1;
foreach $c \in N.\text{fanin}(n)$ **do**
  $\text{ref}(c) \leftarrow \text{ref}(c) - 1$;
  if $\text{ref}(c) = 0$ **then**
    value $\leftarrow$ value + DerefNode(c, L);
return value;

Algorithm 5: RefNode

Data: Boolean network N, node n, leaf set L
Result: Integer value
if $n \in L$ **then return** 0;
value $\leftarrow$ 1;
foreach $c \in N.\text{fanin}(n)$ **do**
  $\text{ref}(c) \leftarrow \text{ref}(c) + 1$;
  if $\text{ref}(c) = 1$ **then**
    value $\leftarrow$ value + RefNode(c, L);
return value;

C. DAG-aware rewriting

DAG-aware rewriting [8] is an effective technique to compute the gain of replacing a part of logic in a Boolean network with another part. The techniques associates each node $n \in N$ with a reference counter, an integer value $\text{ref}(n)$, that keeps track of how many other nodes in N *use* (or depend) on n. Initially, $\text{ref}(n)$ is set to the node's fanout size. An reference counter value $\text{ref}(n) = 0$ denotes that the node is not used by any other logic and can be consequently removed from N.

Removing a node n from the network or adding a node n to the network can be "simulated" by recursively updating the integer values of n and all its predecessors in $\text{TFI}_\infty(n)$ until either the value of a node becomes 0 or a leaf node is reached. Alg. 4 and 5 show how the reference counters are recursively manipulated. The input of the algorithms are a node and a leaf set L.

The gain of replacing a cut $C = (n, L)$ with $C' = (n', L)$ can then be efficiently computed using Algorithm 6. The algorithm is capable of exploiting structural hashing, i.e., nodes already in the Boolean network will not be re-added.

D. Disjoint-support decomposition

A top-down disjoint-support decomposition for a Boolean function $f(x)$, $x = x_1, \ldots, x_n$, checks whether there exists and finds a two-input gate function g, an $(n-1)$-input remainder function f', and a variable index $i \in \{1, \ldots, n\}$ such that

$$f(x) = g(x_i, f'(x_1, \ldots, x_{i-1}, x_{i+1}, x_n)). \quad (1)$$

Similarly, a bottom-up disjoint-support decomposition for f checks whether there exists and finds a 2-input gate function g, an $(n-1)$-input remainder function f', and two distinct variable indexes $i, j \in \{1, \ldots, n\}$ such that

$$f(x) = f'(g(x_i, x_j), x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n). \quad (2)$$

These decompositions "extract" a gate from the original Boolean function, either at the output or at the inputs, thereby reducing the support size by 1, remaining with a smaller function f'.

A disjoint-support decomposition algorithm checks whether an input function f is top-down decomposable or bottom-up decomposable for variable indexes in a fixed given order repeatedly until no further decomposition is possible [14]. The final remainder function f' is called *prime*.

Disjoint-support decomposition can be performed as a preprocessing step to exact synthesis. Every successful decomposition drops one support variable from the original function and already extracts one gate for the final solution. The final prime function is expected to be solved easier without losing the overall optimality guarantee. Further, knowing that the final function $f'(x_1, \ldots, x_{n'})$ is prime and assuming that it depends on all variable in its support, we can lower bound the number of minimum gates to implement f' to its support size n'.

Algorithm 6: Gain

Data: Boolean network N, node n replaced with n', leaf set L
Result: Gain of replacing (n, L) with (n', L) in N
$v_1 \leftarrow \text{DerefNode}(n, L)$;
Insert (n', L) into the network N;
$v_2 \leftarrow \text{RefNode}(n', L)$;
$\text{DerefNode}(n', L)$;
$\text{RefNode}(n, L)$;
return $v_1 - v_2$;

E. Caching strategies

Cut functions in practical Boolean networks are not uniformly distributed at random; instead, it is often the case that many cuts across different Boolean networks have the same cut function. Since Boolean resynthesis—and particularly SAT-based exact synthesis—is a computationally intensive task, it is convenient to store the optimized Boolean network implementation for a given cut function, and reuse it whenever the same cut function occurs. This cache, called *solution cache*, cannot only be used for one (or more) runs of a logic optimization script, but also stored and reused as a database of known solutions that bootstrap an exact synthesis algorithm right from the start. In addition to the solution cache, caching timeouts can further save significant runtime. When a query to an exact synthesis algorithm fails, i.e., no optimum Boolean network could be found for a Boolean function due to a resource limit in the SAT solver, the failed synthesis query is stored in a cache, called *blacklist cache*.

Solution caching. The solution cache stores optimum Boolean networks to exact synthesis problems. If we were to disregard additional divisors that can be used to find the optimum network, the cache can simply be implemented by a hash map that maps Boolean functions to Boolean networks. In order to reduce the size of the cache, the Boolean networks from all entries can be primary outputs in a shared Boolean network, thereby sharing the primary inputs but also some intermediate gates that are present in multiple Boolean networks. Since our implementation also makes use of additional divisors, we need to store them in the cache as well. In this case, the key of the hash map is a tuple $(f, d_1, \ldots, d_k)$, where f is the Boolean function to synthesize and $d_1, \ldots, d_k$ are the functions of the divisors. Inside the shared Boolean network that represents the hash map's values, the divisors are simply represented as additional primary inputs, distinct from the ones that are primary inputs to the optimum Boolean network. The hash map can easily be serialized by storing all Boolean functions as truth tables, e.g., as an array of integers, and some textual representation of the Boolean network, e.g., Aiger for AIGs, or Verilog for more general Boolean networks.

Blacklist caching. For a blacklist cache, there exists no Boolean network as a solution, as the cache should mark functions for which no solution could be found. However, the reason that no solution was found is due to a resource limit, e.g., a conflict limit in the SAT solver. Therefore, a blacklist cache is just a hash set, whose keys are the Boolean function and the exact configuration that was passed to the exact synthesis algorithm. Further, given two configurations, the blacklist cache must define a partial order that can decide whether a configuration is strictly better than another one. Then, the algorithm only needs to retry finding an optimum network for a Boolean function that exists in the blacklist cache, if the new configuration is strictly better than the one in the cache.

IV. Experiments

The resynthesis framework described in the previous sections has been implemented and is available online as a part of the EPFL logic synthesis libraries [12].

Setup of exact DAG-aware rewriting. We have evaluated quality and performance of the framework by realizing an exact DAG-aware rewriting algorithm that combines disjoint-support decomposition with SAT-based exact synthesis and uses caching to cope with the high runtime requirements of SAT-based exact synthesis. As benchmarks, we use the well-known EPFL logic synthesis benchmark suite, which serves as a comparative standard for benchmarking logic optimization algorithms. All experiments have been conducted on an Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz. The correctness of the results has been verified using combinational equivalence checking implemented in ABC [15].

TABLE I
Runtime and node reduction for exact DAG-aware rewriting

Benchmark		**Exact DAG-Aware Rewriting**			
Name	**Size** [-]	**Size** [-]	**Impr** [-]	**Impr**$_{\%}$ [%]	**Runtime** [s]
adder	1020	1011	9	0.88	0.15
bar	3336	3142	194	5.82	0.61
div	57247	57220	27	0.05	3.23
hyp	214335	196081	18254	8.52	20.56
log2	32060	32006	54	0.17	7.06
max	2865	2861	4	0.14	0.66
multiplier	27062	26841	221	0.82	4.12
sin	5416	5404	12	0.22	0.62
sqrt	24618	24615	3	0.01	1.39
square	18484	15347	3137	16.97	2.39
arbiter	11839	11839	0	0.00	6.70
cavlc	693	679	14	2.02	0.44
ctrl	174	131	43	24.71	0.04
dec	304	304	0	0.00	0.00
i2c	1342	1316	26	1.94	0.46
int2float	260	222	38	14.62	0.08
mem_ctrl	46836	46355	481	1.03	14.90
priority	978	970	8	0.82	0.21
router	257	247	10	3.89	0.05
voter	13758	12263	1495	10.87	1.36

The realized DAG-aware exact rewriting algorithm computes for each node in the Boolean network an extended cut limited to at most 10 inputs with additional divisors. Each obtained cut function is first decomposed using disjoint-support decomposition. The remainder functions with up to 6 inputs are synthesized using SAT-based exact synthesis with a conflict limit of 10^5 in the SAT solver. Remainder functions of larger size are discarded. Successful synthesis queries as well as unsuccessful synthesis attempts are cached in the solution and blacklist caches, respectively.

Node reduction and runtime. We run the high-effort resynthesis algorithm four times on all benchmarks while alternating between zero-gain replacements turned on and off. Table I shows that this leads to a substantial node reduction for several of the considered benchmarks. Populated solution and blacklist caches ensure good performance and allow the algorithm to scale even for benchmarks of large size. The table is built as follows: the first two columns present the benchmark name and the initial number of nodes before optimization. The other four columns list the number of nodes after optimization, the absolute improvement in the number of nodes, the relative improvement with respect to the initial benchmark size, and the total runtime in seconds. Overall the algorithm achieves a node reduction of 5.19% in a runtime of 65.03 seconds.

Caching. Table II provides further insight into the caching and the number of occurred resynthesis queries. In total, over all four runs of the algorithm, the resynthesis function has been executed 98179 times with Boolean functions of up to

TABLE II
SUMMARIZED CACHING PARAMETERS OF EXACT SYNTHESIS

Parameter	
Total resynthesis queries	98179
Size of solution cache	57132
Size of blacklist cache	1117

10 variables. The solution cache contains 57132 entries of Boolean functions of up to 6 variables. The blacklist cache lists 117 Boolean functions of up to 6 variables. These Boolean functions could not be synthesized with the given limit of 10^5 SAT conflict. To store both caches (uncompressed), less than 200 MB of memory were required.

V. SUMMARY & CONCLUSIONS

This paper presents a generic resynthesis framework for Boolean networks parameterized in a multi-level logic representation, a cut-computation algorithm, and a resynthesis algorithm. By instantiating this resynthesis framework, an exact DAG-aware rewriting algorithm has been proposed that utilizes a composition of SAT-based exact synthesis, disjoint-support decomposition, and efficient caching strategies to size-optimize Boolean networks. The algorithm follows the recent idea of generic logic synthesis algorithms and, as such, can be applied to different multi-level logic representations, such as AIGs, MIGs or alike, as well as LUT networks.

The framework enables more fine-grained control for high-effort logic synthesis by addressing the high runtime requirements of exact synthesis in various ways. First, disjoint-support decomposition can identify top-down and bottom-up decompositions as trivial cases, which make the exact synthesis problem unnecessarily more difficult. Caches help to store successful as well as unsuccessful exact synthesis attempts, thereby not requiring to find a solution twice, but also only to retry finding solutions for difficult instances if they have not been found before with the same or a weaker configuration.

There are several interesting directions for future work. First, exploiting don't cares of the cut functions can significantly reduce the size of the Boolean network, however, it is non-trivial to store functions with don't care information in the cache. It remains open how to make use of caching and don't care information efficiently at the same time. Second, for DAG-aware rewriting it is beneficial to consider multiple Boolean networks for a cut function, since some solutions may enable a larger amount of sharing with the remaining network. In fact, even non-optimum Boolean networks can lead to a larger gain, if they share more logic with the remaining network. Third, the caching idea can also be used for verification problems, e.g., to store intermediate equivalence checking results; and similar to the proposed method, both for successful and failed attempts.

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-ADG669354 CyberCare, by the EPFL Open Science Fund, and in part by SRC Contracts 2710.001 and 2867.001.

REFERENCES

[1] 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. [Online]. Available: http://dx.doi.org/10.1109/PGEC.1963.263531

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

[3] H. Riener, R. Ehlers, and G. Fey, "CEGAR-based EF synthesis of Boolean functions with an application to circuit rectification," in *Asia and South Pacific Design Automation Conference*, 2017, pp. 251–256. [Online]. Available: https://doi.org/10.1109/ASPDAC.2017.7858328

[4] A. Q. Dao, N. Lee, L. Chen, M. P. Lin, J. R. Jiang, A. Mishchenko, and R. K. Brayton, "Efficient computation of ECO patch functions," in *Design Automation Conference*, 2018, pp. 51:1–51:6. [Online]. Available: https://doi.org/10.1145/3195970.3196039

[5] V. N. Kravets, N. Lee, and J. R. Jiang, "Comprehensive search for ECO rectification using symbolic sampling," in *Design Automation Conference*, 2019. [Online]. Available: https://doi.org/10.1145/3316781.3317790

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

[7] H. Riener, E. Testa, W. Haaswijk, A. Mishchenko, L. G. Amarù, G. De Micheli, and M. Soeken, "Scalable generic logic synthesis: One approach to rule them all," in *Design Automation Conference*, 2019, pp. 70–75. [Online]. Available: https://doi.org/10.1145/3316781.3317905

[8] 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. [Online]. Available: http://doi.acm.org/10.1145/1146909.1147048

[9] H. Riener, W. Haaswijk, A. Mishchenko, G. De Micheli, and M. Soeken, "On-the-fly and DAG-aware: Rewriting Boolean networks with exact synthesis," in *Design, Automation and Test in Europe*, 2019, pp. 1649–1654. [Online]. Available: https://doi.org/10.23919/DATE.2019.8715185

[10] W. Haaswijk, M. Soeken, A. Mishchenko, and G. De Micheli, "SAT-based exact synthesis: Encodings, topology families, and parallelism," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2019. [Online]. Available: https://doi.org/10.1109/TCAD.2019.2897703

[11] L. Stok, "The next 25 years in EDA: A cloudy future?" *IEEE Design & Test*, vol. 31, no. 2, pp. 40–46, 2014. [Online]. Available: https://doi.org/10.1109/MDAT.2014.2313451

[12] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, and G. De Micheli, "The EPFL logic synthesis libraries," *CoRR*, vol. abs/1805.05121, 2018. [Online]. Available: http://arxiv.org/abs/1805.05121

[13] 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. [Online]. Available: http://doi.acm.org/10.1145/296399.296425

[14] V. Bertacco and M. Damiani, "Boolean function representation based on disjoint-support decompositions," in *Int'l Conf. on Computer Design*, 1996, pp. 27–32. [Online]. Available: https://doi.org/10.1109/ICCD.1996.563527

[15] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Int'l Conf. on Computer Aided Verification*, 2010, pp. 24–40. [Online]. Available: https://doi.org/10.1007/978-3-642-14295-6_5