Busy Man's Synthesis: Combinational Delay Optimization With SAT

Mathias Soeken[1] Giovanni De Micheli[1] Alan Mishchenko[2]
[1]Integrated Systems Laboratory, EPFL, Lausanne, Switzerland
[2]Department of EECS, UC Berkeley, CA, USA

***Abstract*—Boolean SAT solving can be used to find a minimum-size logic network for a given small Boolean function. This paper extends the SAT formulation to find a minimum-size network under delay constraints. Delay constraints are given in terms of input arrival times and the maximum depth. After integration into a depth-optimizing mapping algorithm, the proposed SAT formulation can be used to perform logic rewriting to reduce the logic depth of a network. It is shown that to be effective the logic rewriting algorithm requires (i) a fast SAT formulation and (ii) heuristics to quickly determine whether the given delay constraints are feasible for a given function. The proposed algorithm is more versatile than previous algorithms, which is confirmed by the experimental results.**

I. Introduction

Multi-level logic synthesis [1], [2] aims at finding a multi-level network representation for a given Boolean single- or multi-output function. The input function can be given as a truth table, a two-level representation, a binary decision diagram, or a poorly optimized logic network. The goal is to find a logic network of better quality while considering different cost functions. Typical cost functions are, e.g., area (i.e., the number of logic gates), depth (i.e., the size of the critical path), or energy. In many applications, a small depth is of particular interest. Minimizing networks under depth constraints is the focus of this paper.

Logic rewriting [3], [4] based on cut enumeration [5] and k-LUT mapping [6] is one of the most effective optimization techniques for logic networks. k-LUT mapping computes a cover of the logic network with Boolean functions with at most k inputs, called LUTs. k-LUT mapping can be used in technology mapping, e.g., for FPGAs, where one is interested in a cover with few LUTs or a short critical path. But in logic rewriting the LUTs are mapped back into logic networks. Therefore, one is interested in LUTs that optimize the delay or area of the subnetworks.

Lazy man's synthesis (LMS, [7]) is a logic rewriting technique that aims at optimizing depth. The approach is based on two observations: (i) many logic synthesis tools can produce optimal or near-optimal networks, and (ii) larger circuits are composed of smaller ones, which also need to be optimal or near-optimal in order to obtain good global results. LMS first mines these smaller subnetworks by enumerating all cuts [5] in the available designs that have been optimized using state-of-the-art synthesis tools. The subnetworks are recorded in a database, which is used by k-LUT mapping to find replacements for individual k-cuts.

LMS is a fast and effective approach to depth optimization in logic networks. But LMS suffers from several shortcomings. It is practically limited to 6 input functions, since for larger input sizes too many functions exist and the size of the database blows up. For similar reasons, it does not take input arrival times into account nor can it deal with incompletely-specified functions. Finally, separate databases are required when dealing with different elementary gates. Currently databases constructed by LMS are based on two-input gates without XOR and XNOR (in other words, the elementary gates are the same as in And-inverter graphs, AIGs).

We propose a high-effort depth-optimization approach called *Busy Man's Synthesis* (BMS), to overcome these shortcomings. Instead of using a precomputed database, BMS computes optimal subnetworks for k-LUTs on the fly during mapping. BMS finds the delay-optimum networks by taking into account input arrival times that are computed using the current partial mapping. Among the delay-optimum networks, BMS guarantees to find a network with the smallest number of gates thereby also providing network with good area cost.

Clearly, BMS consumes substantial runtime since minimum networks are computed by the SAT solver on the fly. However, computational time is less important if one can achieve provably delay-optimal networks. Two aspects are important to make BMS effective. First, one needs a fast algorithm to find an optimum network quickly if the given delay constraints are realizable. Second, one needs checkers to determine that no network can exist if the delay constraints are too restrictive. We use the SAT solver to implement a fast algorithm that can find an optimum network under realizable delay constraints. Our SAT formulation is based on a formulation to find area-optimum logic networks [8]. We extend it by adding clauses characterizing delay constraints and use CEGAR (counterexample-guided abstraction refinement) to simplify the SAT formulas and reduce solving time. The consistency checkers to prove infeasible delay constraints use structural arguments, such as the maximum number of possible gates in a network and techniques based on functional decomposition. Our experiments show that the BMS outperforms state-of-the-art logic rewriting for depth optimization, including LMS.

II. Preliminaries

A Boolean network for a function of n inputs $x_1, \ldots, x_n$ is a sequence of gates $(x_{n+1}, \ldots, x_{n+r})$ with

$$x_i = x_{j(i)} \circ_i x_{k(i)}, \qquad \text{for } n+1 \leq i \leq n+r. \tag{1}$$

That is, each gate combines two previous gates or inputs with $j(i) < k(i) < i$ using $\circ_i$, which is one of the 16 binary operations [9]. For single-output functions, the last gate x_{n+r} is considered the network's output. For multi-output networks, each gate could potentially be an output. We call a function

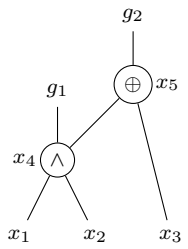

Fig. 1. Example network

normal, if $f(0,\ldots,0) = 0$. A Boolean network represents a normal function if all of its gate functions are normal.

Example 1: Fig. 1 shows an example network with three inputs consisting of two gates: $x_4 = x_1 \wedge x_2$ and $x_5 = x_3 \oplus x_4$. The network has two outputs, $g_1 = x_4$ and $g_2 = x_5$.

We refer to the number of gates r as the *area* of the network. The *delay* is the length of the longest shortest path between primary inputs and primary outputs. The length of a path is measured in terms of the number of gates on the path.

Example 2: The delay of the logic network in the example is 2 as both gates are on the path from x_1 or x_2 to g_2.

We can assume different input arrival times for the inputs of the logic network. This is especially important if we consider subnetworks in the context of a larger network. They influence the overall delay of the network. We refer to δ_i as the input arrival time of input x_i. When computing the delay, the input arrival time needs to be added to the length of the path.

Example 3: Assume $\delta_1 = 0$, $\delta_2 = 0$, and $\delta_3 = 2$. Then, the delay of the network is 3, since now the path from x_3 to g_2 has length 3.

III. Overview of the Algorithm

This section provides an overview of BMS. The algorithm uses dynamic programming to determine the best delay at each node of the subject graph [6], [10]. The nodes are considered in a topological order and the arrival time of at a node is computed assuming the best arrival times at the inputs of a cut are known. To compute the best arrival time at the node, the set of the node's cuts is computed. The cuts are considered in some order, and the best result for each cut is computed. The best result at the node is found as the smallest arrival time possible when using one of the node's cuts.

The Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ of each cut is derived as a truth table and given to the SAT-based minimum-network computation algorithm, together with the best arrival times $\delta_1, \ldots, \delta_n$ of the cut inputs. The algorithm is outlined in Algorithm 1. The SAT solver enumerates different logic structures that can be used to implement the given function and chooses the minimum one that guarantees the minimum arrival time at the output given the arrival times at the inputs.

The entry point of the SAT-based algorithm is procedure 'FindFastestNetwork' which receives as input an n-variable cut function f and input arrival times $\delta = \delta_1, \ldots, \delta_n$. Cut enumeration in the mapping algorithm guarantees that the function functionally depends on all its inputs. The procedure determines the optimum delay by starting from a known-feasible one, referred to as Δ. If the network for a specific delay does not exist, the delay is relaxed and the existence of

```
1  Function FindFastestNetwork(f : B^n → B, δ)
2      set Δ ← ComputeStartDelay();
3      while N' ← FindSmallestNetwork(f, δ, Δ) do
4          set N ← N';
5          set Δ ← Δ − 1;
6      end
7      return N;

8  Function FindSmallestNetwork(f : B^n → B, δ, Δ)
9      if InfeasibleDelay(f, δ, Δ) then return false;
10     foreach r ∈ {n − 1, n, n + 1, ...} do
11         if InfeasibleGates(f, δ, Δ, r) then return false;
12         if N ← FindNetwork(f, δ, Δ, r) then return N;
13     end

14 Function FindNetwork(f : B^n → B, δ, Δ, r)
15     set S ← SATSolver();
16     AddVariables(S, f, δ, Δ, r);
17     if !AddDepthConstraints(S, δ, Δ, r) ∨ !Solve(S) then
18         return false;
19     if !AddGateConstraints(S, f, r) ∨ !Solve(S) then
20         return false;
21     return ExtractNetwork(S);
```

Algorithm 1: Functions to find delay-optimum networks

the networks is checked again. The process continues until a network exists for some delay constraint. The obtained networks are size-optimum, i.e., they require the minimal number of gates for the given delay constraints.

The function 'FindSmallestNetwork' ensures this, which gets as input f, δ, and Δ. Size-optimality is ensured by first checking whether a network exists with $r = n - 1$ gates and then increasing r until a network can be found. However, it may be possible that no network exists that satisfies the delay constraints. Two heuristics apply checks that can find violating constraints: 'InfeasibleDelay' is applied initially once and checks whether the current delay constraints rule out a network; 'InfeasibleGates' is applied in each iteration of the loop. It takes the current number of gates r into account. These checks may not be accurate all the time. They can return false positives, i.e., the check decides that the delay constraints are feasible even when they are not. The checks must not return false negatives, because this will compromise the optimality. Whenever the checks cannot imply that the constraints are inconsistent, the SAT solver is used to check whether there exists a network given r gates.

The function 'FindNetwork' creates a SAT solver, and adds the necessary variables and constraints. Details on how variables and constraints are encoded inside the SAT solver are given in the next section. But at this point we already remark that instead of solving all constraints at once, we first check whether a satisfying assignment can be found only for the depth constraints. These do not have to take f into consideration; and if no satisfying solution can be found for the constraints, the function can exit early. If a satisfying solution is found, the function continues by checking the remaining constraints. Note that the SAT solver may already be able to derive UNSAT while adding constraints.

In order to make Algorithm 1 applicable for practical examples, two requirements are essential: (i) all SAT calls must be fast, i.e., ideally less than a second, which requires effective encodings and/or case-splitting, and (ii) the consistency

checkers must be highly accurate, i.e., they should not generate a lot of false positives as these lead to unnecessary SAT calls. In fact, it is important to stress that both requirements are equally important and build an ideal combination to solve the problem. Without the consistency checkers, the algorithm would spend most of its time in unnecessary SAT solving, which can make the overall algorithm one or two magnitudes slower. Likewise, without the SAT solver one needs to resort to time consuming and ineffective enumerative algorithms to find a network structure that satisfies the given constraints.

IV. Finding Optimum Networks with SAT

A. Knuth's Algorithm

Inspired by the work of Kojevinok et al. [8] and Éen [11], Knuth [12] has proposed a SAT based formulation to find an optimum normal Boolean network for functions $g_1, \ldots, g_m$ depending on n variables. Note that considering normal functions is not restrictive when considering single-output functions. If a function is not normal, we simply find the optimum network for the inverted function and invert the root gate. The idea is to use a SAT solver to check whether there exists a normal Boolean network of r gates that realizes the given functions. This formulation involves variables for indexes $1 \leq h \leq m$, $n < i \leq n + r$, and $0 < t < 2^n$:

$$\begin{aligned} x_{it}: &\quad t^{\text{th}} \text{ bit of } x_i\text{'s truth table} \\ g_{hi}: &\quad [g_h = x_i] \\ s_{ijk}: &\quad [x_i = x_j \circ_i x_k] \text{ for } 1 \leq j < k < i \\ f_{ipq}: &\quad \circ_i(p, q) \text{ for } 0 \leq p, q \leq 1, p + q > 0 \end{aligned} \tag{2}$$

In other words, g_{hi} is true, if function g_h is represented by gate x_i. The variable s_{ijk} is true, if the operands of gate x_i are x_j and x_k. Finally, the variable f_{ipq} is true, if the operation of gate x_i evaluates to true for the input assignment (p, q). Since we consider normal Boolean functions, also each gate maps $(0, 0) \mapsto 0$ which allows us to disregard x_{i0} and f_{i00} for all i.

Example 4: We illustrate the formulation by showing which assignments to the variables represent the Boolean network $x_4 = x_1 \wedge x_2, x_5 = x_3 \oplus x_4$ with $g_1 = x_4$ and $g_2 = x_5$ (see Fig. 1).

The x_{it} variables encode the global function of each gate in terms of a truth table. Since $n = 3$ and $r = 2$, the gate index i ranges from 4 to 5 and there are 7 truth table bits from position 1 to 7.

t	=	7	6	5	4	3	2	1
x_{4t}	=	1	0	0	0	1	0	0
x_{5t}	=	0	1	1	1	1	0	0

There are four g_{hi} variables, of which two are assigned 1 to indicate which output corresponds to which gate.

$$g_{14} = 1,\ g_{15} = 0,\ g_{24} = 0,\ g_{25} = 1$$

There are three variables s_{4jk} and six variables s_{5jk} and from each of these sets of variables one variable is assigned 1.

k	=	2	3	4
s_{41k}	=	1	0	
s_{42k}	=		0	
s_{51k}	=	0	0	0
s_{52k}	=		0	0
s_{53k}	=			1

Finally, the f_{ipq} variables encode the truth tables for the AND and XOR operation.

p, q	=	$1,1$	$0,1$	$1,0$
f_{4pq}	=	1	0	0
f_{5pq}	=	0	1	1

Due to the encoding of the variables, only a few clauses are necessary. The main clauses describe how the truth tables of each gate are computed based on the assignments to the s_{ijk} and f_{ipq} variables. For $0 \leq a, b, c \leq 1$ and $1 \leq j < k < i$ the clauses are

$$\begin{aligned} \big(s_{ijk} \wedge (x_{it} \oplus \bar{a}) \wedge (x_{jt} \oplus \bar{b}) \wedge (x_{kt} \oplus \bar{c})\big) \rightarrow (f_{ibc} \oplus \bar{a}) = \\ (\bar{s}_{ijk} \vee (x_{it} \oplus a) \vee (x_{jt} \oplus b) \vee (x_{kt} \oplus c) \vee (f_{ibc} \oplus \bar{a})) \end{aligned} \tag{3}$$

and have up to five literals. The semantics can readily be observed from the first line in the equation: If gate x_i has x_j and x_k as children, the simulation values for truth table row t at x_j is a, at x_k is b, and at x_i is c, then the function at gate x_i must evaluate to a for the input pair (b, c). Note that a, b, c are constants in (3) and therefore control the polarity of the respective variables. The whole clause or some literals may be omitted in some cases. For example, if $b = c = 0$, we have $f_{i00} = 0$ in the last term. If $a = 0$, the whole clause is omitted, otherwise only the term. Also, x_{jt} and x_{kt} are constants if $j \leq n$ and $k \leq n$, respectively.

Let $t = (t_1 \ldots t_n)_2$ in binary encoding. Then the clauses

$$(\bar{g}_{hi} \vee (\bar{x}_{it} \oplus g_h(t_1, \ldots, t_n))) \tag{4}$$

constrain the output values to the gates they point to. The constraints $\bigvee_{i=n+1}^{n+r} g_{hi}$ ensure that each output is realized by the network and the constraints $\bigvee_{k=2}^{i-1} \bigvee_{j=1}^{k-1} s_{ijk}$ ensure that each gate has two operands.

These constraints are required to make the algorithm work. Additional constraints are not necessary but helpful to reduce the search space for the SAT solver, which brings down the solving time significantly. Further constraints can be used to indicate that each gate is used at least once (i.e., there is no gate with empty fan-out) and that no two gates realize the same operation on the same operands (i.e., the network is structurally hashed). Constraints to break symmetries are also helpful: One can enforce that gates appear in co-lexicographic order if they are independent of each other and one can enforce an order on symmetric variables. The reader is referred to the original reference [12] for the details on these additional constraints.

We integrated the SAT formulation into a CEGAR loop. That is, we don't constrain the whole truth table using the constraints in (3) but check after every satisfying assignment whether the network derived from the assignment is equal to the specification. If that is not the case, we can compute a counter-example which is then used to add clauses for (3) and (4). Eventually, the SAT formulation becomes unsatisfiable implying that there is no network that can realize the input function or the computed network realizes the input function.

B. Encoding Delay Constraints

We extended the synthesis algorithm from the previous section to consider delay constraints and input arrival times. The extension allows for using a SAT solver to check whether

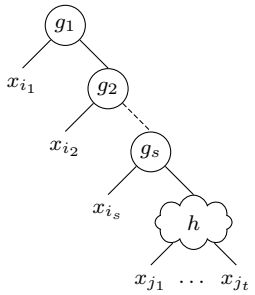

Fig. 2. Stair-decomposition

there exists a Boolean network with r gates that realizes the n-variable functions $g_1, \ldots, g_m$ with a maximum delay of at most Δ assuming the input arrival times $\delta_1, \ldots, \delta_n$. Since logic rewriting is computing the delay and input arrival times based on the logic level, all values are integers.

The idea is to assign a minimum depth to each gate and then constrain the maximum depth of the output gates. To encode the depth at each gate we make use of the *order encoding* [13], [12]. In the order encoding, a value x in the range $0 \leq x \leq M$ is represented by M variables x^l for $1 \leq l \leq M$ where $x^l = [x \geq l]$. We have $x = x^1 + x^2 + \cdots + x^M$, i.e., the bitstring derived by x^j has x ones followed by $(M - x)$ zeros. The clauses

$$(\bar{x}^{l+1} \vee x^l) \tag{5}$$

for $1 \leq l < M$ ensure this property.

Example 5: If $M = 4$, we can represent the values $x = 0, 1, 2, 3, 4$ by the bitstrings $0000, 1000, 1100, 1110, 1111$, respectively.

For the integration of delay constraints into the SAT formulation, we associate a value d_i with each gate to represent a lower bound on the delay of the gate x_i with $n < i \leq n + r$. The value is in the range $0 \leq d_i \leq \delta_{\max} + (i - n)$ where $\delta_{\max} = \max\{\delta_1, \ldots, \delta_n\}$ is the greatest input arrival time. Next, we can encode each d_i in the order encoding using variables d_i^l for $1 \leq l \leq \delta_{\max} + (i - n)$.

We add clauses to propagate the depth limits according to the wiring of the network. Clauses

$$\bigwedge_{l=1}^{\delta_{\max}+j-n} (\bar{s}_{ijk} \vee \bar{d}_j^l \vee d_i^{l+1}) \wedge \bigwedge_{l=1}^{\delta_{\max}+k-n} (\bar{s}_{ijk} \vee \bar{d}_k^l \vee d_i^{l+1}) \tag{6}$$

for $1 \leq j < k < i$ ensure that the minimum delay of gate x_i is larger by at least one compared to the minimum delay of the children, x_j and x_k. For $j \leq n$, the value of d_j^l is the constant value $[\delta_j \geq l]$. The same applies to values of d_k^l for $k \leq n$.

Finally, we add constraints $\bar{g}_{hi} \vee \bar{d}_i^{\Delta+1}$ for primary outputs. If a gate is a primary output, its depth must be less or equal to the maximum depth Δ. Note that this constraint only needs to be added if $\Delta < \delta_{\max} + (i - n)$; otherwise, the maximum depth constraint cannot be violated.

V. Detecting Infeasible Constraints

Since our algorithm to find optimum networks is used in the inner loop of the mapping algorithm, we can make assumptions about the input function f given to 'FindFastestNetwork': f is normal, i.e., $f(0, \ldots, 0) = 0$ and depends on all its inputs.

Input : Function $f : \mathbb{B}^n \to \mathbb{B}$ and indexes $i_1, \ldots, i_s$
Output : True, if f is stair-decomposable w.r.t. indexes
1 set $\mu \leftarrow 2^n - 1$;
2 **foreach** $i \in \{i_1, \ldots, i_s\}$ **do**
3 **switch** IsTopDecomposable(f, μ, x_i) **do**
4 **case** $\wedge$ set $\mu \leftarrow \mu \wedge x_i$;
5 **case** $\vee$, $\overline{\subset}$ set $\mu \leftarrow \mu \wedge \bar{x}_i$;
6 **case** $\oplus$ set $f \leftarrow f \oplus x_i$;
7 **case** *false* **return** *false*;
8 **endsw**
9 **end**
10 **Function** IsTopDecomposable(f, μ, x)
11 **if** $(f \wedge \mu) \subseteq (x \wedge \mu)$ **then return** $\wedge$;
12 **if** $(f \wedge \mu) \subseteq (\bar{x} \wedge \mu)$ **then return** $\overline{\subset}$;
13 **if** $(x \wedge \mu) \subseteq (f \wedge \mu)$ **then return** $\vee$;
14 **if** $(f_x \wedge \mu) = (\bar{f}_{\bar{x}} \wedge \mu)$ **then return** $\oplus$;
15 **return** *false*;

Algorithm 2: Functions to find delay-optimum networks

We check trivial cases, in which $f = 0$ or $f = x_1$ as a preprocessing step. Consequently, f has at least two inputs and therefore the network requires at least one gate.

In the following, we list checks that are performed to detect whether delay constraints are feasible for the given function $f(x_1, \ldots, x_n)$. We first list checks that work independently from the number of gates and only take the input arrival times $\delta_1, \ldots, \delta_n$ and the maximum arrival time Δ into account.

A. Check for Infeasible Delay Constraints

A trivial check is that no network can exist if one of the input arrival times is at least as high as the delay.

Lemma 1: There cannot exist a network for f if $\delta_{\max} \geq \Delta$.

More complicated checks require the use of functional decomposition. We say that f is *stair-decomposable* w.r.t. variables $x_{i_1}, \ldots, x_{i_s}$, if there exists a partition of the input variables $\{x_1, \ldots, x_n\} = \{x_{i_1}, \ldots, x_{i_s}\} \cup \{x_{j_1}, \ldots, x_{j_t}\}$ and functions $g_1, \ldots, g_s, h$ such that

$$f(x_1, \ldots, x_n) = g_1(x_{i_1}, g_2(x_{i_2}, \ldots, g_s(x_{i_s}, h(x_{j_1}, \ldots, x_{j_t})) \ldots)). \tag{7}$$

In other words, f is realized by a network, as depicted in Fig. 2. Note that this network must be optimum for f for any optimum network of h.

Lemma 2: If there are input arrival times $\delta_{i_1}, \ldots, \delta_{i_s}$ such that $\delta_{i_l} = \Delta - l$ for $1 \leq l \leq s$, then a network for f exists only if f is stair-decomposable w.r.t. $x_{i_1}, \ldots, x_{i_s}$.

Example 6: Assume $f(x_1, x_2, x_3, x_4)$ with $\delta_1 = \delta_3 = 0, \delta_2 = 3$, and $\delta_4 = 4$ and $\Delta = 5$. Then, a network for f only exists if it can be decomposed as $g_1(x_4, g_2(x_2, h(x_1, x_3)))$, i.e., f is stair-decomposable w.r.t. x_4 and x_2.

Algorithm 2 checks whether a function f permits a stair-decomposition $x_{i_1}, \ldots, x_{i_s}$. The variables are derived from the input arrival times. For each variable in order, it is checked whether f is top-decomposable. That is, for the first variable, we check whether we can write $f = x_{i_1} \circ g$ for some $\circ \in \{\wedge, \overline{\subset}, \vee, \oplus\}$. (Note that $x_{i_1} \,\overline{\subset}\, g = \bar{x}_{i_1} \wedge g$.) Then, we need to check whether g is top-decomposable w.r.t. the next variable x_{i_2} and so on. However, g may not be completely specified as some of the truth values are masked by x_{i_1}. We capture this by using a mask μ that acts as the care set for g. Also, note that

```
Input  : Input arrival times δ_1, ..., δ_s, delay Δ
Output : upper bound r_max on the number of gates
1  set s ← 1, l ← Δ, a ← n, r_max ← 0;
2  while (s > 0) ∧ (l > 0) ∧ (2a > s) do
3      for i ∈ {1, ..., n} do
4          if δ_i = l then
5              s ← s − 1, a ← a − 1;
6          l ← l − 1;
7          r_max ← r_max + s;
8          s ← 2s;
9      end
10 end
11 return r_max;
```

Algorithm 3: Determining maximum number of gates

instead of using the name g, we use f to refer to the current function in Algorithm 2. The algorithm is illustrated by the following example. Note that we use '$*$' as don't care symbol. For example, instead of writing $f = 1100$ and $\mu = 1010$, we just write $f = 1{*}0{*}$.

Example 7: Let us check with Algorithm 2 whether $f = 0001\,1110\,0000\,0000$ is stair-decomposable w.r.t. x_4, x_3, x_2. The algorithm performs the following checks and updates to f.

$x_4 = 1111\,1111\,0000\,0000 \supseteq f$	$f \leftarrow 0001\,1110\,{*}{*}{*}{*}\,{*}{*}{*}{*}$
$f_{x_3} = \bar{f}_{\bar{x}_3} = 0001\,0001\,{*}{*}{*}{*}\,{*}{*}{*}{*}$	$f \leftarrow 1110\,1110\,{*}{*}{*}{*}\,{*}{*}{*}{*}$
$x_2 = 1100\,1100\,1100\,1100 \subseteq f$	$f = {*}{*}10\,{*}{*}10, {*}{*}{*}{*}\,{*}{*}{*}{*}$

Hence, f is stair-decomposable w.r.t. the given variables.

Further checks are possible using the notion of stair-decomposable functions.

Lemma 3: Let f be stair-decomposable w.r.t. $x_{i_1}, \ldots, x_{i_s}$ with arrival times as in Lemma 2. Also, let there be another input different from x_{i_s} with the same arrival time δ_{i_s}. There can only exist a network for f, if $n = s + 1$.

In this case, the other input represents function h in Fig. 2 and therefore any other input cannot be part of the network.

If f is stair-decomposable and therefore possibly permits some logic network representation, one can find an optimum circuit for h (see Fig. 2) instead for f. Also, h may have a lot of don't care assignments as indicated by Example 3, making the search easier.

Other checks based on functional decomposition are possible, which are only illustrated by means of an example. Let f be a 6-variable function in which the inputs x_1, x_2 and x_3 have input arrival times 4, the other ones less than 4 and we want to find a network with maximum delay 6. Then, f can only be realized if there exists a decomposition $f = h(g_1(x_1, x_2, x_3), g_2(x_4, x_5, x_6))$.

B. Check for Infeasible Gate Constraints

The depth imposes an upper bound on the number of gates: in the worst case f is a binary tree with $2^{\Delta} - 1$ gates. But we can do much better when taking the input arrival times into account. They act as cut-off points in the binary tree.

Example 8: We illustrate the upper bound computation using a 4-variable function f, a maximum delay $\Delta = 5$ and input arrival times $3, 2, 2$, and 0. The full binary tree has 31 gates. But if we cut off branches for the inputs with arrival times 3 and 2 the upper bound reduces to 18 (see Fig. 3(a)).

However, when some inputs are used as cut-off points, fewer inputs remain that have an influence on the maximum number of remaining gates. In the example, the network has four inputs, and three inputs have already been fixed (see Fig. 3(a)), therefore, the number of gates is highly overestimated. Algorithm 3 shows a heuristic for a better estimation based on cut-off points and remaining inputs. In the algorithmic description, s is the number of gates in the current level, l is the current level, and a is the number of available nodes. Whenever an input can be used as a cut-off point (line 4), the number of current gates per level and available nodes is decreased by 1. The number of current gates is accumulated for the maximum number of gates $r_{\max}$ (line 7) and doubled after every iteration (line 8). Algorithm 3 computes six gates as the upper bound for the network in Example 8 (see Fig. 3(b)).

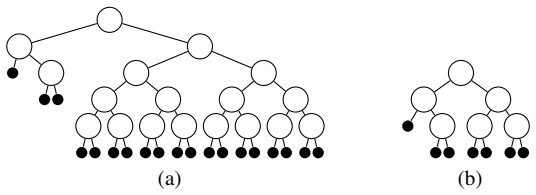

Fig. 3. Maximum number of gates for delay constraints

VI. Experimental Results

This section presents experimental results. We implemented our approach in ABC [15] as command '*if -u*'. Three other depth-optimization commands in ABC are similar to this one: SOP balancing ('*if -g*', [14]), DSD balancing ('*if -x*'), and LMS ('*if -y*', [7]). We applied all four approaches to the combinational ISCAS benchmarks. It is worth noting that LMS's database is constructed using the ISCAS benchmarks (besides other benchmark libraries) and therefore LMS yields near-optimum results for these benchmarks.

Table I lists the results. We used $k = 6$ as cut size. For each delay optimization technique, we list the number of gates and the global depth after optimization as well as the runtime consumed. Each command inputs an AIG and returns a delay-optimized AIG, while our approach returns a network of arbitrary two-input gates, possibly containing XOR gates. Supporting XORs is not only a strength of the algorithm, compared to the other techniques, but is in fact necessary for BMS to be effective. If one permits XOR gates in the SAT formulation (using blocking clauses for the f_{ipq} variables), the runtime decreases significantly in XOR-rich networks. Of course, having XORs in the resulting networks is a main reason for having better depth of the resulting networks. By restricting the SAT formulation to find AIGs, the resulting depth is the same as that computed by LMS, since the database used by LMS is computed using the ISCAS benchmarks.

Some subnetworks are very difficult for the SAT solver to handle. In order to keep the runtimes reasonable, we aborted SAT calls after 400 000 conflicts (see, e.g., [12]). Further, when the call to 'FindNetwork' is aborted in 'FindSmallestNetwork' for some r, we retry to find a network by setting r to the maximum number of gates (see Section V-B) and then decrease r until no network can be found or the search is aborted. This can help find a network, since satisfiability calls are often faster than unsatisfiability calls in these optimization approaches [16]. The procedure may not guarantee size optimality, which is acceptable because we focus on depth optimization.

TABLE I
EXPERIMENTAL RESULTS

Benchmark	SOP balancing [14]			DSD balancing			LMS [7]			BMS				
	gates	depth	runtime	gates	depth	runtime	gates	depth	runtime	gates	depth	runtime	cuts	aborts
c432	202	20	0.01	134	23	0.02	208	20	1.16	211	20	1908.92	7902	4
c499	412	14	0.05	414	15	0.04	414	13	1.26	232	9	5079.43	28509	2
c880	382	14	0.03	322	20	0.04	373	14	1.20	341	14	16213.83	18786	35
c1355	412	14	0.05	414	15	0.04	402	13	1.26	242	9	1894.19	29595	0
c2670	607	15	0.08	581	17	0.08	618	14	1.18	606	14	7617.23	34007	28
c3540	1063	27	0.16	961	32	0.18	1014	27	1.33	1020	22	21297.38	77575	73
c5315	1388	21	0.20	1341	26	0.21	1364	19	1.73	1361	16	133345.26	95124	186
c6288	2792	63	0.11	2874	87	0.32	2684	55	1.87	2274	46	66558.42	163153	355
c7552	1594	17	0.38	1535	25	0.42	1511	17	2.10	1293	16	209267.97	150817	302

Out of the three techniques we used in the comparison, LMS leads to the best results. Therefore, we use LMS as the baseline for comparison in other experiments. The results show that the depth can be further reduced for all but three benchmarks. For these three, *c432*, *c880*, and *c2670*, the depth is the same as obtained by LMS, but in two cases the area is better. Only for *c432* LMS is better than BMS. The best delay improvements are obtained for *c499*, *c1355*, *c3540*, and *c6288*.

BMS as currently implemented is significantly slower than other methods. This is because other approaches consider only a subset of feasible subnetworks, while BMS enumerates through all subnetworks to find a depth-optimum one. However, in order to show the efficiency of BMS despite its high runtime, we report in the last two columns the number of cuts enumerated by the LUT mapper and the number of cuts aborted after hitting the resource limit of 400 000 conflicts. As an example, in *c499* the SAT solver aborted only twice after considering 28 509 cuts. For these two functions, it cannot be guaranteed that the optimum subnetwork has been found, but for the functions of all other cuts this guarantee can be given. In fact, for *c1355* the SAT solver was able to find the exact optimum for all 29 595 enumerated cuts, thereby guaranteeing a local minimum for 6-LUT mapping. This is a strong statement for a logic optimization algorithm.

VII. CONCLUSION

We presented a high-effort depth-reducing logic rewriting algorithm called Busy Man's Synthesis (BMS). The algorithm is implemented on top of a technology mapper. A novel SAT-based method is used to compute a depth-optimal implementations of Boolean functions of k-cuts enumerated by the mapper while taking input arrival times into account. Besides an efficient SAT-based algorithm, high-quality checkers of the feasibility of delay constraints are also essential for a robust implementation. Our experiments show that high-quality results are often produced and local optimality is achieved with enough computational effort—this is a novelty for logic rewriting algorithms. The runtime of the approach grows proportional to the number of cuts, and is therefore also applicable to large networks. Our experiments showed that for 6-input functions the SAT-based algorithm rarely aborted. We expect to scale the algorithm to 7-input and 8-input functions by doubling and quadrupling the resource limit for the SAT solver, respectively.

Future work will include making the algorithm more efficient by exploiting satisfiability don't cares, which can be computed by the mapper for Boolean functions of each cut, and by using NPN classification to reduce the number of functions evaluated by the SAT solver. Further, we need to strengthen the constraint feasibility checkers using functional decomposition or by computing better upper bounds.

Acknowledgments: This research was partly supported by the European Research Council (H2020-ERC-2014-ADG 669354 CyberCare), the Swiss National Science Foundation (200021_169084 MAJesty), and the NSF/NSA grant "Enhanced equivalence checking in cryptoanalytic applications" at University of California, Berkeley.

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] P. Bjesse and A. Borälv, "DAG-aware circuit compression for formal verification," in *International Conference on Computer-Aided Design*, 2004, pp. 42–49.
[4] 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.
[5] P. Pan and C.-C. Lin, "A new retiming-based technology mapping algorithm for LUT-based FPGAs," in *FPGA*, 1998, pp. 35–42.
[6] J. Cong and Y. Ding, "FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 13, no. 1, pp. 1–12, 1994.
[7] W. Yang, L. Wang, and A. Mishchenko, "Lazy man's logic synthesis," in *Int'l Conf. on Computer-Aided Design*, 2012, pp. 597–604.
[8] A. Kojevnikov, A. S. Kulikov, and G. Yaroslavtsev, "Finding efficient circuits using SAT-solvers," in *Theory and Applications of Satisfiability Testing*, 2009, pp. 32–44.
[9] D. E. Knuth, *The Art of Computer Programming, Volume 4A*. Reading, Massachusetts: Addison-Wesley, 2011.
[10] Y. Kukimoto, R. K. Brayton, and P. Sawkar, "Delay-optimal technology mapping by DAG covering," in *Design Automation Conference*, 1998, pp. 348–351.
[11] N. Éen, "Practical SAT – a tutorial on applied satisfiability solving," 2007, slides of invited talk at FMCAD.
[12] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Reading, Massachusetts: Addison-Wesley, 2015.
[13] J. M. Crawford and A. B. Baker, "Experimental results on the application of satisfiability algorithms to scheduling problems," in *National Conf. on Artificial Intelligence*, 1994, pp. 1092–1097.
[14] A. Mishchenko, R. K. Brayton, S. Jang, and V. N. Kravets, "Delay optimization using SOP balancing," in *Int'l Conf. on Computer-Aided Design*, 2011, pp. 375–382.
[15] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Computer Aided Verification*, 2010, pp. 24–40.
[16] N. Lodha, S. Ordyniak, and S. Szeider, "A SAT approach to branchwidth," in *Theory and Applications of Satisfiability Testing*, 2016, pp. 179–195.