SAT-Based Exact Synthesis: Encodings, Topology Families, and Parallelism

Winston Haaswijk, *Student Member, IEEE*, Mathias Soeken, *Member, IEEE*, Alan Mishchenko, *Senior Member, IEEE*, and Giovanni De Micheli, *Fellow, IEEE*

***Abstract*—Exact synthesis is a versatile logic synthesis technique with applications to logic optimization, technology mapping, synthesis for emerging technologies, and cryptography. In recent years, advances in SAT solving have led to a heightened research effort into SAT-based exact synthesis. Advantages of exact synthesis include the use of various constraints (e.g., synthesis of emerging technology circuits). However, although progress has been made, its runtime remains unpredictable. This paper identifies two key points as hurdles to further progress. First, there are open questions regarding the design and implementation of exact synthesis systems, due to the many degrees of freedom. For example, there are different CNF encodings, different symmetry breaks to choose from, and different encodings may be suitable for different domains. Second, SAT-based exact synthesis is difficult to parallelize. Indeed, this is a common drawback of logic synthesis algorithms. This paper proposes four ways to close some open questions and to reduce runtime: 1) quantifying differences between CNF encoding schemes and their impacts on runtime; 2) demonstrating impact of symmetry breaking constraints; 3) showing how directed acyclic graph topology information can be used to decrease runtime; and 4) showing how topology information can be used to leverage parallelism.**

***Index Terms*—Boolean satisfiability, circuit synthesis, design automation, exact synthesis, logic synthesis, SAT.**

I. Introduction

EXACT synthesis is a term used by the logic synthesis community for any method that can be applied to yield *exact* results for logic synthesis problems. In this context, the term exact synthesis is not used in opposition to approximate synthesis, which is a paradigm concerned with the synthesis of systems that produce approximately correct results [1]. Rather, exact synthesis refers to synthesizing logic representations that *exactly meets a specification*. For example, given a Boolean function $f : \mathbb{B}^n \to \mathbb{B}^m$ and an number $r \in \mathbb{N}$ we may ask

$\mathcal{Q}_1$: "Does there exist a logic network N such that N implements f with exactly r gates?"

or

$\mathcal{Q}_2$: "Does there exist a sum-of-product (SOP) expression E with exactly r cubes that represents f?"

An exact synthesis algorithm can be used to answer such questions. We are interested in constructive algorithms such that, if a question $\mathcal{Q}_x$ can be answered in the affirmative, we want to know a logic representation that meets the specification. In the above examples, we want our algorithm to produce a logic network N or an SOP expression E.

The notion of exactness is closely related to that of *optimality*. Given an algorithm for the exact synthesis of some representation form, we can often adapt it to synthesize optimum representations. Suppose, we have a constructive algorithm for $\mathcal{Q}_1$. We could then use it to synthesize size-optimum logic networks as follows. Initialize r to zero and query the algorithm. Increment r until we find the first value r' for which the algorithm reports success. This r' must then be the size of the smallest, i.e., size-optimum, logic network for f. Due to the close correspondence between exact- and optimum synthesis, the terms are often used interchangeably. In fact, the term exact synthesis is widely used to refer to the synthesis of optimum representations.

Exact synthesis algorithms exist for both two- and multilevel logic representations. The Quine–McCluskey algorithm and Petrick's method are well-known algorithms for the minimization of SOPs [2], [3]. Similar methods have been developed for so-called *exclusive* SOPs as well [4]. In multilevel logic synthesis, we encounter various exact minimization algorithms, such as the decomposition techniques of Ashenhurst [5], Curtis [6], Davidson [7], and Roth and Karp [8]. More recently, enumeration-based techniques have been developed by Knuth [9] and Amarú *et al.* [10]. In practice, heuristic methods are often preferred for performance reasons [11]. The heuristic counterparts to two-level exact synthesis are the espresso and exorcism algorithms [12], [13]. For multilevel logic, algebraic and Boolean methods exist [12].

Exact synthesis has practical as well as theoretical applications. Practical applications range from logic optimization, technology mapping, and synthesis for emerging technologies to less obvious ones, such as cryptography [14]–[19]. On the theoretical side, it allows us to derive upper and lower bounds on the complexity of functions [20]. It is known that all 4-variable Boolean functions can be represented using SOPs with at most eight implicants [21]. Using exact synthesis, Knuth has found that all 5-variable Boolean functions can be

Manuscript received August 27, 2018; revised November 20, 2018; accepted January 10, 2019. Date of publication February 4, 2019; date of current version March 18, 2020. This work was supported in part by Semiconductor Research Corporation Contract 2710.001 and Contract 2867.001 through SAT-Based Methods for Scalable Synthesis and Verification at UC Berkeley under Grant H2020-ERC-2014-ADG 669354 CyberCare, and in part by the Swiss National Science Foundation under Grant 200021-169084 MAJesty. This paper was recommended by Associate Editor J. Cortadella. *(Corresponding author: Winston Haaswijk.)*

W. Haaswijk, M. Soeken, and G. De Micheli are with the Integrated Systems Laboratory, EPFL, 1015 Lausanne, Switzerland (e-mail: winston.haaswijk@epfl.ch).

A. Mishchenko is with the Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720 USA.

Digital Object Identifier 10.1109/TCAD.2019.2897703

represented using 2-input gate-level networks with at most 12 gates [9]. In this paper, rather than concentrating on applications, we show improvements to the core exact synthesis algorithm.

In recent years, significant strides have been made in algorithms for Boolean satisfiability (SAT) [22]. These developments, coupled with increases in compute power, have led to a resurgence of exact synthesis algorithms backed by SAT solver backends [15]–[17]. Despite this progress, its adoption has been limited, due to its unpredictable runtime. There have been attempts to mitigate runtime with techniques, such as the development of alternative CNF encodings, the addition of symmetry breaking clauses, and the use of *counterexample-guided abstraction refinement* [9], [23]. However, these techniques are often applied in an ad-hoc matter. Moreover, it is not clear how the various encodings and constraints interact with different SAT solvers. To date no comprehensive quantitative comparison of the various methods exists. This presents difficulties in the design of new systems, as there is no data to use as a basis for any design choices. Another hurdle is that, like many EDA algorithms, it is difficult to parallelize. Some efforts have been made in parallelizing SAT solvers using techniques, such as *cube-and-conquer*, clause sharing, and *portfolio* SAT solvers which apply different SAT solvers in a parallel or distributed manner [24], [25]. This has proven difficult, partially due to theoretical limitations of the resolution procedure [26]. Moreover, solvers based on these methods are typically domain agnostic, and do not take advantage of specific domain structure.

Our contributions can be divided into three parts.

1) We present a series of experiments which demonstrate, for the first time, quantitative differences between CNF encodings. These results can be used as a basis for the design and implementation of SAT-based exact synthesis systems. The experiments are implemented with the open source *percy* tool, which is available to the public at https://github.com/whaaswijk/percy.
2) We introduce novel algorithms based on families of directed acyclic graph (DAG) topologies. We show that they can be used to reduce synthesis runtime as well as the number of timeouts (thus increasing the number of solved instances).
3) We show how topology information can be used to transform the SAT-based exact synthesis problem into an embarrassingly parallel one. This allows us to design parallel algorithms that are up to 68× faster than the state-of-the-art.

The rest of this paper is organized as follows. Section II formalizes Boolean networks and provides background on finding optimum Boolean networks using SAT-based exact synthesis.

The first part of our contributions starts in Section III, where we describe and measure in detail three different CNF encodings and symmetry breaking constraints.

Next, in Section IV, we describe two different types of DAG topology families. We discuss some of their theoretical properties, algorithms for generating them, as well as how they can be used to improve synthesis runtime.

Then, in Section V, we show how topology families can be used to unlock parallel synthesis algorithms, and provide some experimental results that show their performance.

Finally, we conclude this paper with a brief discussion in Section VI.

II. Background

In this section, we describe the background of (generalized) Boolean chains and SAT-based exact synthesis. Both of these concepts will be used extensively throughout the text.

A. Boolean Chains

We present here an extension of Boolean chains, a concept originally introduced by Knuth [9]. Knuth's formalization is limited to chains consisting of 2-input operators. Here, we extend this definition to k-input operators, where k is arbitrarily large, but fixed.

A Boolean chain is a DAG in which every vertex corresponds to a k-input Boolean operator $\phi : \mathbb{B}^k \to \mathbb{B}$. Following the convention of [8], we denote the set of allowed operators by $\mathcal{B}$. Boolean chains are compact structures for the representation of multiple-output Boolean functions, similar to the concept of *unbound logic networks* used by the logic synthesis community [11], although they are slightly more restricted. Their formal definition is as follows. Let $f = (f_1, \ldots, f_m)$ be a multiple-output Boolean function, such that $f : \mathbb{B}^n \to \mathbb{B}^m$ and the functions $f_1, \ldots, f_m$ are defined over common support $x_1, \ldots, x_n$. Then, for $k \geq 1$ and a set $\mathcal{B}$, a k-input operator Boolean chain is a sequence $(x_{n+1}, \ldots, x_{n+r})$, where

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

such that $\phi_i \in \mathcal{B}$, $1 \leq j(i,\cdot) < i$, and for all $1 \leq k \leq m$, either $f_k(x_1, \ldots, x_n) = x_{l(k)}$ or $f_k(x_1, \ldots, x_n) = \bar{x}_{l(k)}$, where $0 \leq l(k) \leq n+r$, and $x_0 = 0$ the constant zero input. For example, in Knuth's definition of Boolean chains, $\mathcal{B}$ is the set of all binary operators. The objects $x_{n+1}, \ldots, x_{n+r}$ are called the *steps* of the chain.

For example, when $n = 3$, then the 2-input operator 5-step chain

$$\begin{aligned} x_4 &= x_1 \wedge x_2 \\ x_5 &= x_1 \oplus x_2 \\ x_6 &= x_3 \wedge x_5 \\ x_7 &= x_3 \oplus x_5 \\ x_8 &= x_4 \vee x_6 \\ l(1) &= 7 \\ l(2) &= 8 \end{aligned}$$

can be used to represent the 3-input 2-output function $f(x_1, x_2, x_3) = (x_1 \oplus x_2 \oplus x_3, \langle x_1, x_2, x_3 \rangle)$, which is commonly known as a full adder.[1] Fig. 1 illustrates this example.

The extension of Boolean chains to arbitrary k-input operators has several motivations. First, synthesis of chains with larger operator sizes may be significantly faster. For example, using 3-input operator Boolean chains, one can efficiently classify the set of all 5-input functions using SAT-based exact synthesis [27], whereas this has not been achieved for 2-input operator chains. Second, one application of exact synthesis is in technology mapping, where we are often required to use a diverse set of logic *primitives*. Generally, we cannot assume that a given cell library contains only 2-input

[1] We use angular brackets to denote the majority function.

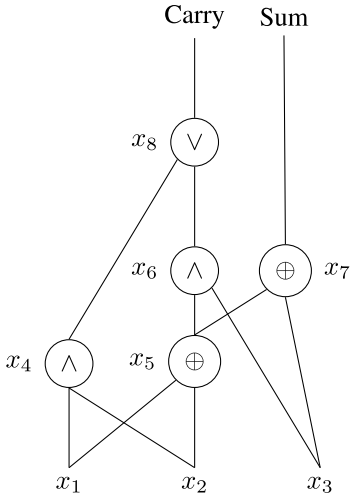

Fig. 1. Illustration of a normal 2-input operator Boolean chain for a full adder. This chain also happens to be a size-optimum. As it is not used, the constant zero input x_0 is not shown here.

operators. Finally, recently there has been a resurgence of bounded logic network representations, such as MIGs and XMGs [16]. These require operators ranging from 3 to at least 5 inputs, although this depends on the specific representation (i.e., we typically understand MIGs to require 3-input operators).

We say that a Boolean chain is *normalized* or *normal* if all it steps correspond to normal functions, i.e., functions that output zero when all of their k inputs are zero. For example, a chain consisting of AND and OR functions is normal, but a chain of NANDs is not.

B. SAT-Based Exact Synthesis

The first example of SAT-based exact synthesis that we are aware of is the tutorial on "Practical SAT" given by Eén at the FMCAD conference [28]. Later, Kojevnikov, Kulikov, and Yaroslavtsev used an extended CNF encoding to find circuit-size upper bounds [29]. Later, Knuth implemented his own formulation which uses a somewhat different CNF encoding and was limited to 2-input operator chains [30]. These algorithms all aim to find size-optimum Boolean chains. Soeken *et al.* [17] extended them to synthesize depth-optimum chains instead. In this paper, our focus is on methods for size-optimum synthesis but, due to the large overlap in methodology, the results should carry over to the depth-optimum case as well.

The principle idea behind these methods is the same. Given a function $f : \mathbb{B}^n \to \mathbb{B}^m$, they do the following.

1) Initialize $r \leftarrow 0$.
2) Encode $\mathcal{Q}_1$ as a CNF formula $\mathcal{F}_r$.
3) Feed $\mathcal{F}_r$ to an SAT solver and wait for its result.
4) If the result is SAT then we are done. An optimum-size chain can be extracted from the satisfying solution.
5) Otherwise, the result is UNSAT. In this case we set $r \leftarrow r + 1$ and go to step 2.

Hence, the size-optimum problem can be solved by a sequence of SAT formulas. This process is captured by Fig. 2.

We are free to choose between distinct (but equivalent) CNF encodings $\mathcal{F}_r$. However, it may not be clear which one is best in a given context.

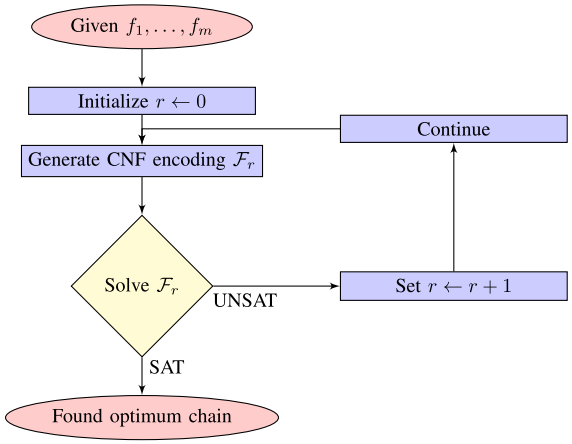

Fig. 2. Illustration of a size-optimum SAT-based exact synthesis algorithm.

C. Note on Optimality

It seems prudent here to address a point of confusion which sometimes arises when discussing optimum synthesis. We always refer to optimality within the context of a specific model of computation. The model of computation used throughout this paper is that of Boolean chains. Suppose, we synthesize a function f and obtain the chain C. When we say that C is size-optimum, this means that there exists no chain C' that computes f with fewer steps than C. That is not to say that there may not exist different models of computation, such as cyclic combinational circuits [31], in which f could be implemented with fewer computational primitives.

III. Analysis of CNF Encodings

In this section, specifically Sections III-A–III-C, we describe three different CNF encodings. This is not meant to be an exhaustive list. Other encodings exist, including the one proposed by Kojevnikov *et al.* [29]. Rather, we present these encodings as they are heavily used in practice, and yet we are unaware of any detailed descriptions or comparisons in existing literature. Section III-D describes a number of symmetry breaking constraints which can be used to speed up synthesis. In Section III-E, we show, for the first time, a comprehensive comparison between the encodings, including their behavior under various symmetry breaking constraints.

In the following, we assume the generic synthesis problem in which we are given the multiple-output Boolean function $f = (f_1, \ldots, f_m) : \mathbb{B}^n \to \mathbb{B}^m$, and we wish to synthesize a 2-input operator Boolean chain. While all encodings we describe can be used for the synthesis of k-input operator chains, for clarity we describe only the 2-input case. The extension to arbitrary k is then straightforward.

A. Single Selection Variable Encoding

The single selection variable (SSV) encoding is typically used for the synthesis of *normal* 2-input operator chains. The normalization requirement does not limit the optimality of synthesized chains: any function computed by a non-normalized chain can be computed by a normalized chain with the same number of steps. One can simply complement the desired non-normal function, synthesize a normal chain, and invert it. The use of normal chains has the advantage that they can be built

out of normal steps. This reduces the number of variables needed by the encoding.

In this encoding, $\mathcal{F}_r$ consists of the following variables, for $1 \leq h \leq m$, $n < i \leq n + r$, and $0 < t < 2^n$:

$$\begin{aligned} x_{it} &: t^{\text{th}} \text{ bit of } x_i\text{'s truth table} \\ g_{hi} &: f_h(x_1, \ldots, x_n) = x_i \\ s_{ijk} &: x_i = x_j \circ_i x_k \text{ for } 1 \leq j < k < i \\ f_{ipq} &: p \circ_i q \text{ for } 0 \leq p, q \leq 1, p + q > 0. \end{aligned}$$

Here, the g_{hi} variables determine which outputs point to which step. The s_{ijk} variables determine the inputs j and k, for each step i. These are also known as *selection variables*. The f_{ipq} encode for all steps i what the corresponding Boolean operator is. Note that we do not encode f_{i00}, since $f_i(0, 0) = 0$ by definition of normal chains.

These variables are then constrained by a set of clauses which ensures that the chain computes the correct functions. For $0 \leq a, b, c \leq 1$ and $1 \leq j < k < i$, the main clauses are

$$\big(\bar{s}_{ijk} \vee (x_{it} \oplus a) \vee \big(x_{jt} \oplus b\big) \vee (x_{kt} \oplus c) \vee (f_{ibc} \oplus \bar{a})\big).$$

Intuitively, these clauses encode the following constraint: if step i has inputs j and k *and* the tth bit of x_i is a *and* the tth bit of x_j is b *and* the tth bit of x_k is c, *then* it must be the case that $b \circ_i c = a$. This can be understood by rewriting the formula as follows:

$$\big(\big(s_{ijk} \wedge (x_{it} \oplus \bar{a}) \wedge \big(x_{jt} \oplus \bar{b}\big) \wedge (x_{kt} \oplus \bar{c})\big) \rightarrow (f_{ibc} \oplus \bar{a})\big).$$

Note that a, b, and c are constants which are used to set the proper variable polarities.

Let $(b_1, \ldots, b_n)_2$ be the binary encoding of truth table index t. In order to fix the proper output values, we add the clauses $(\bar{g}_{hi} \vee \bar{x}_{it})$ or $(\bar{g}_{hi} \vee x_{it})$ depending on the value $f_h(b_1, \ldots, b_n)$. Next, for each output, we add $\bigvee_{i=n+1}^{n+r} g_{hi}$. This ensures that one of its corresponding output variables must be true. In other words, it ensures that every output points to a step in the chain. Finally, for each step, we add $\bigvee_{k=1}^{i-1} \bigvee_{j=1}^{k-1} s_{ijk}$. This ensures that one of its selection variables must be true. Particularly, it ensures that every step in the chain has two valid fanins.

Let us consider the full-adder chain in Fig. 1 as an example. Since, in this case $n = 3$ and $r = 5$, we have to encode the following truth table bits:

$$\begin{array}{rccccccc} t = & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ x_{4t} = & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ x_{5t} = & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ x_{6t} = & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ x_{7t} = & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ x_{8t} = & 1 & 1 & 1 & 0 & 1 & 0 & 0. \end{array}$$

There are two outputs and each of them may be connected to exactly one step. Since there are five steps, we have a total of ten g_{hi} variables to encode all possible output connections. Two of these are set to one to indicate which steps correspond to outputs: $g_{17} = g_{28} = 1$. All other g_{hi} are zero.

Similarly, from the DAG structure of the network, we can see that $s_{412} = 1$, $s_{512} = 1$, $s_{635} = 1$, $s_{735} = 1$, and $s_{846} = 1$. All other s_{ijk} are zero.

Finally, the variables encoding the Boolean operators are assigned to the following values:

$$\begin{array}{rccc} (p, q) = & (1, 1) & (0, 1) & (1, 0) \\ f_{4pq} = & 1 & 0 & 0 \\ f_{5pq} = & 0 & 1 & 1 \\ f_{6pq} = & 1 & 0 & 0 \\ f_{7pq} = & 0 & 1 & 1 \\ f_{8pq} = & 1 & 1 & 1. \end{array}$$

This variable assignment satisfies all clauses and the chain that computes the full adder can be extracted from the CNF formula simply by inspecting the selection and operator variables.

A key difference between the encodings in this section is in the number of s_i variables, also known as the *selection variables*, that they use. Let us therefore compute the number of selection variables in the SSV encoding. All possible operand pairs for step i are explicitly encoded by separate variables $s_{ijk} (j < k < i)$. For a given i there are $\binom{i-1}{2}$ possible operand pairs to choose from. Thus, the total number of selection variables in the SSV encoding is

$$\sum_{i=n+1}^{n+r} \binom{i-1}{2} = \frac{1}{6}\Big(3n^2 + 3n(r-2) + r^2 - 3r + 2\Big).$$

In other words, it is quadratic in the number of inputs n and gates r.

B. Multiple Selection Variables Encoding

In the multiple selection variable (MSV) encoding, we define the following variables $1 \leq h \leq m$, $n < i \leq n + r$, and $0 < t < 2^n$:

$$\begin{aligned} x_{it} &: t^{\text{th}} \text{ bit of } x_i\text{'s truth table} \\ g_{hi} &: f_h(x_1, \ldots, x_n) = x_i \\ s_{ij} &: x_i \text{ has operand } j \text{where } 1 \leq j < i \\ f_{ipq} &: p \circ_i q \text{ for } 0 \leq p, q \leq 1, p + q > 0. \end{aligned}$$

The MSV encoding uses the variable s_{ij} to indicate that step i has operand j. Thus, it requires only $i - 1$ selection variables per step. The total number is

$$\sum_{i=n+1}^{n+r} (i - 1) = \frac{1}{2}(2n + r - 1).$$

Thus, the MSV encoding reduces the number of variables from a quadratic to a linear number, as compared to the SSV encoding. However, it achieves this reduction in variables at the cost of additional clauses. It must maintain the cardinality constraint that $\sum_{j=1}^{i-1} s_{ij} = 2$. In this case that constraint can no longer be enforced by a single clause. One solution is to add the clauses

$$\bigwedge_{j<k<l<i} \big(\bar{s}_{ij} \vee \bar{s}_{ik} \vee \bar{s}_{il}\big)$$

and

$$\bigwedge_{k=1}^{i-1} \big(s_{i1} \vee \cdots \vee s_{i(k-1)} \vee s_{i(k+1)} \vee \cdots \vee s_{i(i-1)}\big).$$

Intuitively, such clauses work as follows. They state that in any triplet of potential operands for step i at least one must be false. Moreover, consider a set of operands which consists of all potential operands of i with one removed. In such a set at least one operand must be used by i. Thus, by adding this second set of clauses we ensure that at least two operands are used. Combined, these constraints therefore ensure that exactly two operands are selected. The drawback of these constraints is that they require

$$\sum_{i=n+1}^{n+r} \binom{i-1}{3} + \binom{i-1}{i-2}$$

additional clauses, which is quadratic in n and r.

Fortunately there exist more efficient encoding schemes. One example is to add a unary binary counter (UBC) circuit to the CNF. Essentially such a circuit acts as a (partial) ripply carry adder which allows us to ensure that the total number of selected operands is equal to 2. Moreover, it uses only a linear number of clauses. Finally, it has the advantage that as soon as two operands are selected, the entire circuit is computed by unit propagation, exploiting the SAT solver's efficiency. A complete description of this circuit is outside the scope of this paper, but we refer the interested reader to [32]. We use the UBC encoding in all our experiments.

After putting the appropriate cardinality constraints in place, for $0 \leq a, b, c \leq 1$ and $1 \leq j < k < i$, the main clauses are now

$$\left(\bar{s}_{ij} \vee \bar{s}_{ik} \vee (x_{it} \oplus a) \vee \left(x_{jt} \oplus b\right) \vee (x_{kt} \oplus c) \vee (f_{ibc} \oplus \bar{a})\right).$$

Similar to the SSV encoding, we add the clauses $(\bar{g}_{hi} \vee \bar{x}_{it})$ or $(\bar{g}_{hi} \vee x_{it})$ depending on the value $f_h(t_1, \ldots, t_n)$. We also add $\bigvee_{i=n+1}^{n+r} g_{hi}$.

Example: Let us consider again the previous example of encoding the full-adder. It is similar to the SSV encoding, with the only difference being in the selection variables. We now have

$$\begin{aligned} s_{41} &= s_{42} = 1 \\ s_{51} &= s_{52} = 1 \\ s_{63} &= s_{65} = 1 \\ s_{73} &= s_{75} = 1 \\ s_{84} &= s_{86} = 1 \end{aligned}$$

and all other s_{ij} zero.

C. Distinct Input Truth Tables Encoding

The distinct input truth tables (DITT) encoding possesses some interesting structural differences from the previous two. In the SSV and MSV encodings, there is a tight coupling between the selection variables and the propagation of truth table bits through the operator variables. The DITT encoding removes that direct coupling at the cost of introducing additional variables and clauses. However, while it creates more variables, it simultaneously reduces the complexity of the clauses.

Let us begin by defining the variables

x_{it} : t^{th} bit of x_i's truth table
$x_{it}^{(k)}$: t^{th} bit of x_i's k^{th} input truth table, $k \in \{1, 2\}$
$g_{hi} : f_h(x_1, \ldots, x_n) = x_i$
$s_{ij}^{(k)}$: Input k of x_i has operand j for $1 \leq j < i$, $k \in \{1, 2\}$
$f_{ipq} : p \circ_i q$ for $0 \leq p, q \leq 1, p + q > 0$.

The output and operator variables are equivalent to those in the previous encodings. The difference lies in the selection variables and propagation of truth table bits. Previously, we defined t truth table bit variables for each step. In this case, we define the additional variables $x_{it}^{(k)}$ which correspond to the truth tables of the inputs to step i. The actual values of those bits depend on which inputs i has selected. In this encoding, we define selection variables for each fanin of a step. Variables for the different fanins are indexed by k, whose range depends on the operator size (2 in this case). Obviously this encoding requires more variables. For example, it encodes three times as many truth table bits. However, it recovers this complexity by reducing the complexity of constraints.

The main clauses are now

$$\left((x_{it} \oplus a) \vee \left(x_{it}^{(1)} \oplus b\right) \vee \left(x_{it}^{(2)} \oplus c\right) \vee (f_{ibc} \oplus \bar{a})\right).$$

Note the structural difference with the above encodings here. In those, the main clauses combine the selection variables and the truth table bits to propagate truth table and operator bits. The DITT essentially removes this coupling. Instead, the structure-based propagation of truth table bits is determined by adding the clauses $s_{ij}^{(k)} \rightarrow (x_{it}^{(k)} = x_{jt})$. In other words, the input truth table bits (used in the main clause) are now determined directly by the selection variables.

Finally, we ensure that all step fanins point to some input by adding $\bigwedge_{k=1}^{2} \bigvee_{j=1}^{i-1} s_{ij}^{(k)}$.

Let us count the number of selection variables used in this encoding. Consider a step x_i. Each of its k fanins may select any of the previous $i-1$ steps. Therefore, the number of selection variables per step is $k(i-1)$. The total number of selection variables for all steps is then

$$\sum_{n+1}^{n+r} k(i-1) = k \sum_{n+1}^{n+r} (i-1) = \frac{k}{2}(2n + r - 1).$$

Thus, we require k times as many selection variables as in the MSV encoding. However, the number is still linear in n and r.

There is another subtle difference between this encoding and the previous two. In fact, the DITT encoding is more general. It allows step fanins to be ordered arbitrarily: the kth fanin of step i may point to step $i' + m$ ($m > 0$), even when fanin $k + 1$ points to step i'. This flexibility allows it to synthesize a larger class of logic networks as compared to the previous encodings. Those only synthesize Boolean chains which can be viewed as a logic network in which gate fanins are ordered tuples. Although this flexibility may be desirable in some cases, it also increases the search space. Therefore, in the context of synthesis for Boolean chains, we add the additional clauses $\bigwedge_{j=1}^{i-2} \bigwedge_{j'=1}^{j} (\bar{s}_{ij}^{(1)} \vee \bar{s}_{ij'}^{(2)})$ to ensure that all step fanins are ordered.

D. Symmetry Breaking

The encodings as we have described them so far are sufficient to synthesize any Boolean chain. Here, we briefly describe several optional *symmetry breaking* clauses. These clauses are not required to produce correct results, but may be

used to constrain the SAT solver's search space while still providing exact results. As such, the aim of adding these clauses is to reduce runtime at the cost of additional clauses and CNF encoding complexity. Due to this additional cost, it is *a priori* not clear how they affect synthesis runtime. In Section III-E, we present a number of experiments to elucidate their impact. These constraints are due to Kojevnikov *et al.* [29] and Knuth [30]. We describe them here using the SSV encoding for 2-input chains, but it is straightforward to generalize this descriptions to other encodings and input sizes.

*1) Only Nontrivial Operands (**N**):* Any optimum Boolean chain will not contain any trivial Boolean operands, such as variable projections or the constant 1 and 0 functions. We may exclude these by adding the additional clauses $(f_{i01} \vee f_{i10} \vee f_{i11})$, $(f_{i01} \vee \bar{f}_{i10} \vee \bar{f}_{i11})$, and $(\bar{f}_{i01} \vee f_{i10} \vee \bar{f}_{i11})$.

*2) Use All Steps (**A**):* An optimum chain must use all its steps to compute its output value (otherwise we could remove the unused steps). To enforce this constraint, we can add the clauses

$$\left(\bigvee_{k=1}^{m} g_{ki} \vee \bigvee_{i'=i+1}^{n+r} \bigvee_{j=1}^{i-1} s_{i'ji} \vee \bigvee_{i'=i+1}^{n+r} \bigvee_{j=i+1}^{i'-1} s_{i'ij} \right)$$

for all i.

*3) No Reapplication of Operands (**R**):* Adding the clauses $(\bar{s}_{ijk} \vee \bar{s}_{i'ji})$ and $(\bar{s}_{ijk} \vee \bar{s}_{i'ki})$ for $i < i' \leq n + r$ ensures that the chain never *reapplies* an operator. Intuitively, suppose that step i has inputs j and k. If $i' > i$ has inputs j and i (or k and i) then step i is redundant: i' may as well act on inputs j and k directly (since steps can implement arbitrary 2-input operators).

*4) Co-Lexicographically Ordered Steps (**C**):* Without loss of generality, we may impose a co-lexicographical order on the step fanins. In other words, a step like $x_7 = \circ_7(x_3, x_4)$ need never follow a step $x_6 = \circ_6(x_2, x_5)$. We can do this by adding $(\bar{s}_{ijk} \vee s_{(i+1)j'k'})$ if $j' < j < k = k'$ or if $k' < k$.

*5) (Co-)Lexicographically Ordered Operands (**O**):* Similarly to the previous point, we may enforce an order on step operators as well. We can do this by adding the clauses $((s_{ijk} \wedge s_{(i+1)jk}) \rightarrow f_i \prec f_{(i+1)})$. In this case, we are free to choose a lexicographic or co-lexicographic order, depending on the relation $\prec$.

*6) Ordered Symmetric Variables (**S**):* If two function inputs p and q are symmetric ($p < q$), we may ensure that input p is used before q. To do so, we can add the clauses

$$\left(\bar{s}_{ijq} \vee \bigvee_{n<i'<i} \bigvee_{1 \leq j' < k' < i'} \left[j' = p \text{ or } k' = p \right] s_{i'j'k'} \right)$$

whenever $j \neq p$.

E. Quantitative Comparison of CNF Encodings

Now that the various encodings and symmetry breaks are defined, we are in a position to perform the experiments in which we compare them. We would like to be able to answer the following questions about encodings.

1) Which is fastest on representative benchmarks.
2) What is the impact of various symmetry breaks.
3) Does (1) change when we increase operator size.

The answer to question 3) tells us if some encodings are better suited for different step operator sizes. This is related to domain suitability, as different domains may require different operator sizes. For example, when synthesizing or mapping into arbitrary-input MIGs, we may wish to use a synthesis engine that is well suited for the synthesis of large operators, whereas this is not the case for AIG synthesis [16], [33].

TABLE I
IMPACT OF SYMMETRY BREAKING ON THE SPACE OF 4-INPUT FUNCTIONS FOR 2-INPUT OPERATOR CHAINS. SORTED BY AVERAGE SYNTHESIS TIME. ALL TIMES REPORTED IN ms

Enc	Symmetries						Avg. time	Stdev	Worst time	#V/C
	N	A	R	C	O	S				
SSV	1	1	1	1	0	1	97.43	261.08	2,150.39	0.2/6
SSV	1	1	1	1	1	1	106.90	266.19	1,880.79	0.2/7
MSV	1	1	1	1	0	1	139.50	376.34	2,704.15	0.3/6
MSV	1	1	1	1	1	1	155.48	407.59	2,472.43	0.3/7
DITT	1	1	1	1	0	1	182.19	493.07	3,570.65	0.3/6
DITT	1	0	1	1	0	1	188.73	515.51	3,522.34	0.3/6
DITT	0	0	0	0	1	0	967.22	3,014.48	18,452.16	0.3/4
DITT	0	1	0	0	1	0	1,019.73	3,164.27	19,694.84	0.3/4
MSV	0	0	0	0	1	0	1,262.76	4,106.57	24,730.40	0.3/6
SSV	0	0	0	0	0	0	1,280.86	4,172.96	28,247.10	0.2/5
MSV	1	0	0	0	1	0	1,281.85	4,062.07	22,593.47	0.3/6
SSV	0	0	0	0	1	0	1,414.26	4,738.83	30,136.69	0.2/6

In our first experiment, we synthesize size-optimum 2-input operator Boolean chains for all 222 4-input NPN classes. We do so using all three encodings and all 2^6 possible symmetry breaking settings. In other words, for each encoding, we try all possible combination of symmetry breaks, on all 4-input functions. The results of this experiment are summarized in Table I, where we have selected, for each encoding, the two best and the two worst settings with respect to average synthesis runtime. In the symmetries column, a 1 (0) means that a symmetry break was enabled (disabled).

Table I shows average synthesis runtime, standard deviation, worst case runtime, as well as the average number of variables and clauses (in thousands) in satisfiable CNF formulas. Note that, in this experiment, it is important to control for the time spent generating the encoded CNF formulas. Some encoders may be faster than others up to some constant factor which depends on implementation details. However, we are interested in the merits of the encodings themselves. In other words, we want to compare the difficulty of solving the different CNF formulas and *not* the time taken by some specific implementation to generate them. In practice, good encoder implementations are fast and time spent encoding is negligible: the asymptotic behavior of the synthesis algorithm is determined heavily by the CNF. Therefore, we consider encoding time as noise and measure only time spent by the SAT solver. Moreover, we synthesize each function twice and measure the average runtime, so as to further reduce noise-induced variance. This experiment, and all the following ones, was executed on a machine with a 2× Intel Xeon E5-2680 v3 processor with a 30 MB cache and 256 GB DDR4-2133 RAM.

First, let us consider the impacts of symmetry breaking. The results show that symmetry breaks have a very significant impact on runtime. For example, the best SSV encoding enables most symmetry breaks and is 14.5× faster than the worst, which disables almost all of them. We see similar behavior for the MSV and DITT encodings as well. Their best settings are more than 9× and 5.5× faster than their worst settings, respectively. Next, let us look at the differences between

TABLE II
IMPACT OF ENCODING AND SYMMETRY BREAKING FOR 5-INPUT FUNCTIONS WITH 3-INPUT OPERATOR CHAINS. TIMES IN ms

Enc	Symmetries						Avg. time	Stdev	Worst time	#V/C
	N	A	R	C	O	S				
MSV	1	1	1	1	0	0	2,422.89	1,650.90	14,356.44	.4/24
MSV	1	1	1	1	0	1	2,429.79	1,682.05	14,357.73	.4/24
SSV	0	1	1	1	0	1	3,153.26	2,742.46	29,278.75	.4/27
SSV	0	1	1	1	0	0	3,218.43	2,817.81	29,310.33	.4/27
MSV	0	1	0	0	1	1	3,756.41	2,059.06	17,370.34	.4/24
MSV	0	1	0	0	1	0	3,877.60	2,334.24	17,382.18	.4/24
SSV	1	0	0	0	1	1	6,729.75	3,117.73	23,056.81	.4/27
SSV	1	0	0	0	1	0	6,748.37	3,054.30	23,063.86	.4/27
DITT	1	0	1	0	0	1	8,354.00	8,998.35	66,569.10	.7/13
DITT	1	0	1	0	0	0	8,711.28	10,723.85	99,439.82	.7/13
DITT	1	0	0	1	1	1	18,265.28	23,652.56	16,5613.63	.7/224
DITT	1	0	0	1	1	0	20,234.97	33,959.06	35,1973.54	.7/224

TABLE III
IMPACT OF ENCODING AND SYMMETRY BREAKING FOR 6-INPUT FUNCTIONS WITH 4-INPUT OPERATOR CHAINS. TIMES IN ms

Enc	Symmetries						Avg. time	Stdev	Worst time	#V/C
	N	A	R	C	O	S				
MSV	1	1	1	0	0	1	74.87	225.65	2,272.33	0.4/25
MSV	1	1	1	0	0	0	77.23	235.32	2,292.44	0.4/25
DITT	1	0	0	0	0	1	78.37	297.98	3,102.35	1.0/16
DITT	0	0	0	0	0	1	79.81	314.68	3,096.09	1.0/16
SSV	0	1	1	0	0	1	87.29	336.08	3,494.17	0.4/27
SSV	0	1	1	0	0	0	89.79	353.71	3,490.85	0.4/27
SSV	1	1	1	1	1	0	118.23	431.94	4,584.01	0.4/31
SSV	0	0	0	1	1	0	120.74	418.12	3,422.77	0.4/30
MSV	1	1	0	1	1	0	128.16	340.11	2,635.04	0.4/28
MSV	1	1	0	1	1	1	126.27	346.01	2,633.60	0.4/28
DITT	0	0	0	1	1	0	5,588.13	18,06.48	124,664.75	1.0/4968
DITT	1	0	0	1	1	0	5,629.10	18,247.72	126,259.26	1.0/4968

encodings. The best SSV encoding is 30% and 47% faster than the best MSV and DITT encodings, respectively. Thus, we see that the choice of encoding and symmetry breaks has a notable impact on synthesis runtime.

In our next experiment, we investigate question 3) by measuring runtime while increasing the number of inputs as well as Boolean chain operator size. Therefore, we now synthesize 5-input functions using Boolean chains with 3-input operator steps. The space of 5-input functions is too large to run this experiment on all of them. Instead, we synthesize 222 randomly sampled 5-input functions. Table II contains the summary of results.

We now find the MSV encoding to be the fastest. It is 23% and 3.4× faster than the best SSV and DITT encodings, respectively. Furthermore, symmetry breaking settings again make a significant difference, with difference of 2.1×, 1.6×, and 2.4× between the best and worst SSV, MSV, and DITT encodings, respectively.

To further investigate the impact of different encodings on input and operator scaling, we test on a set of 500 non-DSD decomposable 6-input functions. These functions were harvested from the MCNC/ISCAS/ITC benchmark suites and should therefore be representative of functions which appear in concrete circuits. We now perform synthesis for chains with 4-input operators. Such large operators are used in (re-)synthesis and mapping of k-LUTs. Results are reported in Table III.

We see that the MSV and DITT encodings are now both starting to outperform the SSV one. They are 14% and 10% faster, respectively. This is likely caused by the selection variable scaling described above. As the chain operator size increases, so do the number of possible fanin combinations. Since the number of selection variables in the MSV and DITT encodings scales linearly, we expect these encodings to be more efficient than the SSV one, which scales quadratically. Again, there are significant differences between the best and worst symmetry breaking settings of encodings. The runtime difference is 1.7×, 72×, and 28% for the MSV, DITT, and SSV encodings, respectively.

The experiments clearly show that the choice of encoding and symmetry breaks has a great impact on the expected runtime. The best choice depends heavily on both the function domain and operator size. Runtime differences between different encodings can be significant (up to 3.5×), but the largest impact is due to symmetry breaking within encodings (up to 72×). Interestingly, enabling more symmetry breaks does not guarantee improved runtimes.

TABLE IV
NUMBER OF FUNCTIONS WITH GIVEN NUMBER OF STEPS

Nr. of steps	4-input functions	5-input functions	6-inputs functions
0	2	0	0
1	2	0	0
2	5	0	110
3	20	2	384
4	34	122	6
5	75	98	0
6	72	0	0
7	12	0	0

Table IV shows the size distributions of the functions synthesized in Tables I–III. We can see, for example, that the maximum number of 2-input operator steps required for a 4-input function is 7. The size distribution for 5-input functions with 3-input operators follows the one found in [27], with the maximum number of steps now being 5. This is as expected when randomly sampling functions. Finally, as one would expect, when operator size increases, the number of required steps decreases accordingly.

IV. DAG TOPOLOGY FAMILIES

SAT-based synthesis always has to contend with unpredictable, and potentially slow runtimes. This is perhaps unsurprising if we consider that, in finding optimum Boolean chains, the SAT solver has to simultaneously perform at least two distinct tasks.

1) Finding valid DAG structures for the Boolean chain.
2) Assigning Boolean operators to the vertices in these DAGs, such that the entire sequence of the chain corresponds to the specified Boolean function.

It has been known for some time that when a solver is supplied with a valid DAG structure the synthesis problem is greatly simplified. Suppose we are given a DAG $G = (V, E)$, and a Boolean function $f : \mathbb{B}^n \to \mathbb{B}^m$. We may be able to transform the DAG into a Boolean chain for f by assigning the appropriate operators $\phi_i \in \mathcal{B}$ to every vertex $v_i \in V$. We call such a transformation a *labeling* of the graph. Finding such a labeling may not be possible, but if it exists, an SAT solver can find it efficiently. For example, consider the single-output 6-input

function with truth table 0x9ef7a8d9c7193a0f.[2] The smallest known implementation of this function uses 19 2-input gates. When a solution topology is given, an SAT solver can find a labeling in 0.12 s on a laptop computer. Without this topology, finding a solution is intractable. The above solution was obtained using a combination of Boolean decomposition and circuit enumeration.

The efficiency of labeling may inspire one to think of a (naive) synthesis algorithm which, given f, simply enumerates DAG structures until it finds one that can be labeled. Such an algorithm reduces to efficiently finding a DAG with the proper structure for f. However, in general, given f we do not know *a priori* which DAG structures have a labeling. Given an n-input function, finding a suitable DAG requires us to search a very large space of DAG structures. Unfortunately, the enumeration of potential DAGs in this space generally outweighs the potential efficiency of graph labeling. To see why, we can refer to the first column of Table V, which contains the numbers of DAGs up to 12 vertices.

Alternatively, we can specify a set of clauses which constrain the SAT solver's search to a particular family of DAG topologies. We then use the SAT solver's efficient search heuristics to find only those topologies within that family. This approach avoids explicit enumeration of DAGs and provides a middle ground between the unstructured exact synthesis formulation of Sections II-B and III on the one hand, and the fully structured labeling of graphs on the other hand. In Sections IV-A and IV-B, we introduce two different types of topology families. Both explore this middle ground in different ways and can be used to achieve significant runtime improvements over conventional unstructured encodings.

A. Fences

Given two integers k and l ($1 \leq l \leq k$), a *Boolean fence* is a partition of k nodes over l levels, where every level contains at least one node. We can denote a Boolean fence by an ordered sequence $F = (\lambda_1, \ldots, \lambda_l)$, where every λ_i corresponds to the collection of nodes on level i. A Boolean fence (k, l) is not unique: there may be multiple ways of distributing k nodes over l levels. We call the set of all such partitions a Boolean fence *family* and write $\mathcal{F}(k, l)$. We use $\mathcal{F}_k$ to denote the set of all fence families of k nodes

$$\mathcal{F}_k = \{\mathcal{F}(k, l) \mid 1 \leq l \leq k\}.$$

To be concise, we also refer to Boolean fences and fence families as fences and families, respectively. Boolean fences can be visualized as graphs. Fig. 3 shows the fences in $\mathcal{F}_4$.

Every DAG of n nodes corresponds to a unique fence $F \in \mathcal{F}_n$. To see why, note that we can assign levels to nodes in a DAG based on their partial order. Such an assignment allows us to find the level distribution corresponding to the fence F.

A fence induces a set of DAG topologies, in which each topology corresponds to the same distribution of nodes over levels, but with different arcs between nodes. In other words, fences represent families of graph topologies. Consequently, a fence induces a set of Boolean chains with those topologies.

[2]For conciseness, we represent the binary truth table as a hexadecimal string, where the right-most characters represent the least significant bits.

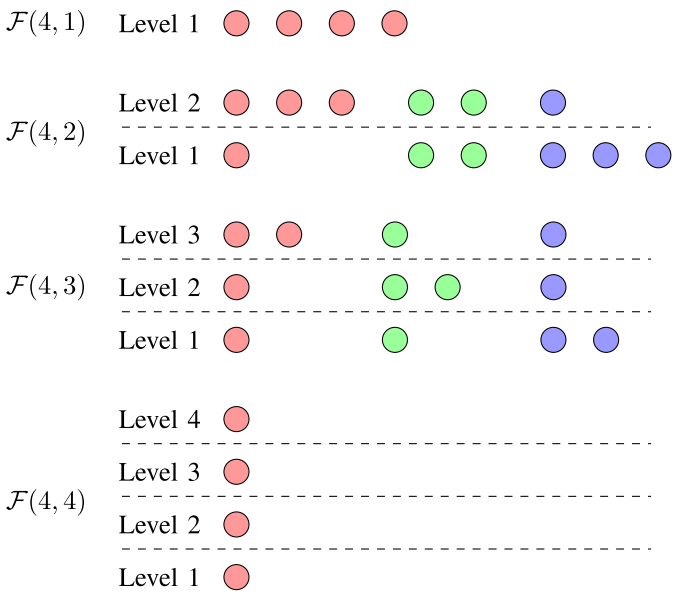

Fig. 3. Illustration of the fences in $\mathcal{F}_4$. Every fence corresponds to a family of DAGs with the same distribution of nodes across levels.

B. Partial DAGs

Fences are one type of topology family which can be used to add some additional structure to SAT-based exact synthesis. However, they still leave a fair bit of structure unspecified. For instance, they do not specify any connections between steps. Moreover, they are even agnostic with respect to the number of possible fanins of each node. In some scenarios this flexibility may be desirable. However, in others we might benefit from additional structure. For instance, we may know that we want to synthesize Boolean chain with 2-input operators up to some number r steps. Preferably, our synthesis method would be able to take advantage of this information.

A *partial DAG* is a topological structure which may be viewed as a partial specification of the underlying DAG structure for a Boolean chain. It specifies two things: 1) the number of fanins for each step and 2) the connections between internal nodes. All connections to primary inputs are left unspecified. Note that one can recover a level distribution from the internal connections of a partial DAG. Hence, partial DAGs contain more structural information than fences.

More formally, a partial DAG of n nodes can viewed as a sequence of k-steps

$$(x_{11}, x_{12}, \ldots, x_{1k}), \ldots, (x_{n1}, x_{n2}, \ldots, x_{nk}).$$

If $x_{ij} = 0$ ($j < i$), then the jth fanin of step i points to some unspecified primary input. Otherwise, if $x_{ij} = m$ ($m < i$), then the jth fanin of step i points to the mth step in the chain. Fig. 4 shows an example of a partial DAG and the corresponding sequence of steps. Note that, like fences, partial DAGs are agnostic with respect to the number of primary inputs they should be synthesized with.

We can efficiently generate (and filter) partial DAGs through a recursive backtrack search algorithm, similar to a fence-generating algorithm. Additionally, we can perform SAT-based exact synthesis using partial DAGs in a similar way to fence-based synthesis, reducing the size of CNF formulas through the structural information encoded in the DAGs.

TABLE V
COMPARING THE NUMBERS OF DAGS, PARTIAL DAGS, AND FENCES FOR INCREASING NUMBERS OF VERTICES

Nr. of vertices	DAGs	Unfiltered PDs/2	Filtered PDs/3	Filtered PDs/2	Fences	Fences 1/3	Fence 1/2
1	1	1	1	1	1	1	1
2	3	2	1	1	2	1	1
3	25	8	3	3	4	2	2
4	543	56	15	9	8	4	3
5	29,281	616	45	41	16	7	6
6	3,781,503	9,856	383	235	32	14	12
7	1,138,779,265	216,832	3,512	1,660	64	28	23
8	783,702,329,343	6,288,128	33,696	13,961	128	56	45
9	1,213,442,454,842,881	232,660,736	344,691	136,875	256	112	90
10	4,175,098,976,430,598,143	10,702,393,856	3,701,536	1,536,631	512	224	180
11	31,603,459,396,418,917,607,425	599,334,055,936	41,204,800	19,484,561	1,024	448	360
12	521,939,651,343,829,405,020,504,063	40,155,381,747,712	472,131,247	275,949,886	2,048	895	719

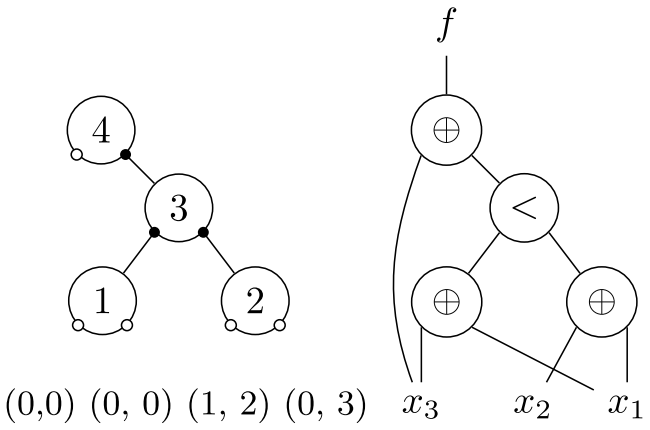

Fig. 4. On the left an example of partial DAG specified by the sequence below. Unspecified fanins are signified by empty circles. On the right a fully specified chain found by the SAT solver for the function $f = \langle x_1 x_2 x_3 \rangle$. The $<$ operator is defined as $< (x_1, x_2) = \bar{x}_1 x_2$.

C. Counting Dags, Fences, and Partial DAGs

Let us consider the following question: how many fences are there in family $\mathcal{F}(k, l)$? Note that, in this family, l nodes are fixed, since we need to have at least one node on l levels. The remaining $k-l$ nodes may be arbitrarily distributed across the l levels. In other words, our question reduces to: how many ways are there to distribute $k-l$ indistinguishable nodes across l bins? The answer is equal to the number of non-negative integer-valued solutions to the equation

$$x_1 + x_2 + \cdots + x_l = k - l$$

and hence

$$|\mathcal{F}(k, l)| = \binom{k-1}{l-1}. \tag{1}$$

We can now use (1) to count the total number of fences of k nodes, $|\mathcal{F}_k|$ as follows:

$$|\mathcal{F}_k| = \sum_{i=1}^{k} \binom{k-1}{i-1} = 2^{k-1}.$$

The reader may verify that these formulas correctly predicts the numbers of fences in Fig. 3. This formula for the number of fences confirms our intuition. Although the number of fences grows exponentially, it is still many orders of magnitude less than the number of DAGs (see Table V). Moreover, there are some other techniques we can use to reduce the number of fences that are "relevant" to a given synthesis problem. For instance, if we want to synthesize a single-output function, we may disregard all fences that have more than one node on the top level. Similarly, if we know that the operators in chain we want to synthesize have fanin 2, we may disregard fences that have more than two nodes directly below the top level. Through this process, which we call *filtering* we can further reduce the number of fences that we need to consider. In Table V, we show the number of fences needed for the common problems of synthesizing single-output functions for chains with 2- and 3-input operators. We write Fences x/y to signify the number of filtered fences relevant to x-output functions and chains with y-input operators.

Counting the number of partial DAGs is slightly more involved as it depends on the fanin size k. We show here a derivation for the number of partial DAGs with fanin size 2. Obviously, there is only 1 partial DAG with 1 node. It consists of the single step sequence $(0, 0)$ since the node may only point to primary inputs. In a partial DAG with two nodes, the second node may either point to two primary inputs, or select a primary input and the first node. Similarly, a third node could either point to two primary inputs, or select a primary input and the first node, a primary input and the second node, or select both preceding steps. From the pattern that arises we can see that generally the nth node has $1 + \binom{n}{2}$ possible fanin options: either it has two primary input fanins, or it may select two distinct fanins from the n-element set of previous steps and primary inputs. Therefore, the possible number of n step partial DAGs F_n is given by the formula

$$F_n = \prod_{i=1}^{n} \left(1 + \binom{i}{2}\right)$$

where we follow the convention that $\binom{1}{2} = 0$.

Table V shows the number of partial DAGs up to 12 nodes (Unfiltered PD/2). We write PD/k for the number of partial DAGs with k-fanin steps. While the number of partial DAGs is orders of magnitude smaller than the total number of DAGS, it is still quite large. Fortunately, we can perform a number of filtering steps. For example, we may use some of the symmetry breaks described in Section III to reduce the number of DAG topologies. Furthermore, for any set of isomorphic partial DAG topologies, we may select one representative and remove the others. We use the Nauty package to efficiently find isomorphic partial DAGs [34]. Here, we are helped by the

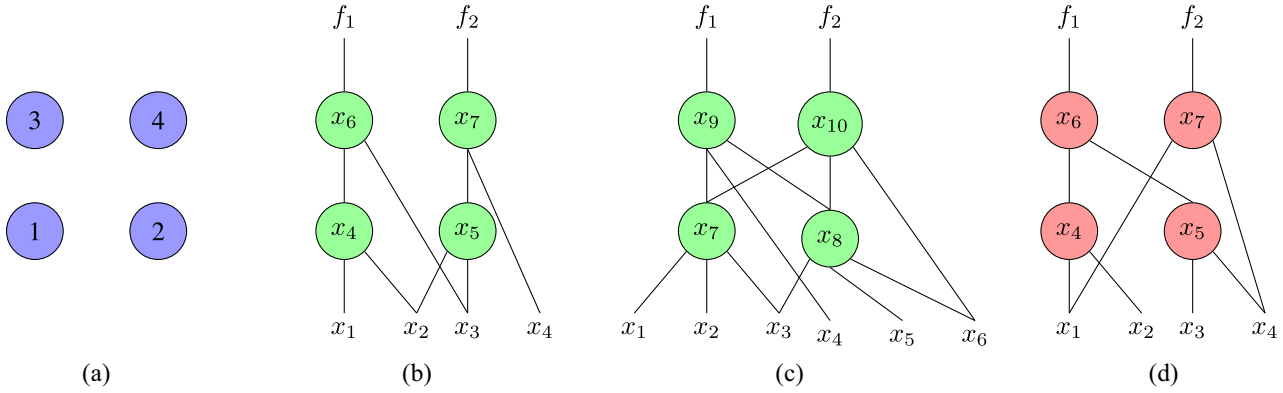

Fig. 5. Fence F in (a) corresponds to a set of possible DAG topologies and can thus be used to constrain the SAT solver's search. For instance, (b) and (c) satisfy the constraints from F while (d) does not. Each node on level λ must have at least one fanin from level $\lambda - 1$; this follows by definition of levels.

fact that all nodes in an n node partial DAG with k-steps have bounded degree. We can find isomorphisms between DAGs of bounded degree in polynomial time [35]. Table V also shows the number of filtered partial DAGs for 2-steps and 3-steps. These numbers are again orders of magnitude smaller than the total number of partial DAGs (of 2-steps, and 3-steps, respectively). Indeed, the numbers are small enough that they may be kept in memory, stored on disk, or in a database. When compressed all the partial DAGs up to 12 nodes for 2-steps take up less than 1GB of space.

D. Generating Fences

As we have seen, fences are simple combinatorial structures which are easy to count. It is therefore perhaps unsurprising that generating them is also simple and can be done efficiently. There exist algorithms based on integer partitioning or backtracking which can be used to efficiently generate streams of fence structures. For a detailed description of one such algorithm, we refer the interested reader to [36].

E. Exact Synthesis Using Fences

We have seen how fences correspond to families of DAG topologies, investigated some of their theoretical properties, and presented a fence generating algorithm. In this section, we consider how to use fences to accelerate exact synthesis by using them to provide additional constraints in the SAT formulation. To do so, let us first look at some connections between fences and Boolean chains.

Consider a fence $F = (\lambda_1, \ldots, \lambda_l)$. Let $G = (V, E)$ be a DAG, and let $\tau(v) : V \to \mathbb{N}$ be the function that assigns each vertex from G to its level. Let $\tau_i = |\{v \mid \tau(v) = i\}|$. We say that G *satisfies* F if and only if $|\lambda_i| = \tau_i$. In other words, a DAG satisfies the topological constraints of a fence if its distribution of nodes across levels is the same. We say that a Boolean chain satisfies F if its underlying DAG structure satisfies F. We consider the primary inputs of the chain to have level 0, and do not consider them in satisfying F.

For example, consider the fence $F = (\lambda_1, \lambda_2) \in F(4, 2)$ highlighted in Fig. 5(a). We have numbered its nodes to make them easier to distinguish. Intuitively, only DAGs with two nodes on the first level and two nodes on the second level satisfy F. For example, Fig. 5(b) is a 2-input operator Boolean chain satisfying the constraints from F. Similarly Fig. 5(c) is a 3-input Boolean chain that satisfies F. However, Fig. 5(d) shows a chain that is invalid for F. It violates the constraint that the step corresponding to fence node 4 be on level 2.

Observe that the topology constraints captured by fences are independent of number of inputs, or operator fanin. This is desirable, as it implies that the same fence generator can be used as the basis for synthesis of generalized Boolean chains and functions of arbitrary input size.

Now consider again the arbitrary fence $F = (\lambda_1, \ldots, \lambda_l) \in F(k, l)$. Suppose we wish to synthesize a Boolean chain that satisfies F. We know that it must be a k-step chain. We assign step x_i to level t by setting

$$\tau(x_i) = t \Leftrightarrow t = \min_{t'} i \leq \sum_{j=0}^{t'} |\lambda_j|$$

where $|\lambda_0| = n$, the number of primary inputs.

Note that if $\tau(x_i) = t$, then step x_i must, by definition, have at least one fanin on level $t-1$. Thus, the fence constrains not only the distribution of nodes across levels, but also the fanin relations between nodes. Due to this level constraint, in the SAT formulation the selection variable s_{ijk} may never be true if $\tau(k) < t - 1$, for any $i < k$. Let k' and k'' be the smallest and largest indices such that $\tau(x_{k'}) = t - 1$ and $\tau(x_{k''}) = t - 1$, respectively. A simple way to express the constraints imposed by the fence is by adding, for each step x_i, the clause $\bigvee_{k=k'}^{k''} s_{ijk} (j < k)$. In that way, we ensure that each step has at least one fanin from a level directly below. This approach is similar to the way that colexicographic or other symmetry-breaking clauses are added in [30]. However, we can do better. As none of the variables outside of $\{s_{ijk} \mid k' \leq k \leq k''\}$ may be true, we do not need to include them in our SAT formula at all. Thus, with fence we can significantly reduce the number of variables and clauses in our SAT instances.

To implement exact synthesis with topological constraints, we can then proceed as follows.

1) Generate a new fence using some fence-generating algorithm.
2) Using the constraints implied by the fence, generate a reduced SAT formula. We use a set of clauses analogous to the one described in Section II-B. However, we exclude any variables or clauses that are rendered unnecessary due to the fence constraints, obtaining a simpler SAT formula.

TABLE VI
COMPARING FENCE- AND PARTIAL DAG-BASED SYNTHESIS TO CONVENTIONAL STATE-OF-THE-ART ENCODINGS. ALL RUNTIMES IN ms

Benchmark	SSV			Fence			Partial DAG		
	mean	#timeouts	#ok	mean	#timeouts	#ok	mean	#timeouts	#ok
NPN4	225.46	0	222	216.69	0	222	75.40	0	222
FDSD6	69.00	0	1,000	29.61	0	1,000	82.41	0	1,000
PDSD6	43,453.33	256	744	20,707.11	128	872	3,613.25	5	995
FDSD8	5,583.13	0	100	2,688.51	0	100	31,379.47	0	100
PDSD8	150,533.31	42	58	100,871.79	11	89	131,625.42	84	16

3) If the formula is satisfiable, we are done.
4) Otherwise, go to 1).

If we incrementally increase the size of the fences that are generated this procedure is guaranteed to find a size-optimum chain. Thus, we extend the conventional exact synthesis algorithm, while decomposing the search space using families of graph topologies. Recall that in Section IV-C we derived the total number of fences of k nodes. Given an upper bound on the number of nodes to realize a function, we therefore also have an upper bound on the number of decomposed exact synthesis instances we have to solve.

F. Fence Versus Conventional Encodings

To evaluate the performance of our proposed approach, we measure the runtimes of different exact synthesis encodings on the following collections of Boolean functions.

1) *NPN4:* All 222 4-input NPN classes [37].
2) *FDSD6:* 1000 fully DSD decomposable 6-input functions that occur frequently in practical synthesis and technology mapping applications [38].
3) *PDSD6:* 1000 common 6-input *partially*-DSD functions.
4) *FDSD8:* 100 fully DSD decomposable 8-input functions.
5) *PDSD8:* 100 partially DSD decomposable 8-input.

We compare three different encodings to synthesize 2-input operator chains for these sets of functions.

1) *SSV:* A baseline implementation of the SSV encoding described in Section III. We enable all symmetry breaks described there, as we experimentally found that this works best for the synthesis of 2-input operator chains.
2) *Fence:* Our proposed algorithm based on fence enumeration and the use of additional topological constraints.
3) *Partial DAGs:* Our algorithm based on partial DAGs.

Table VI lists the results. For each approach three values are listed: 1) the mean solving time (*mean*) in milliseconds; 2) the number of instances that could not be solved in under 3 min (*#t/o*); and 3) the number of instances that were successfully solved within the timeout limit (*#ok*). Note that the number of solved instances is the most important metric here, as it captures in essence how practical an algorithm is. Given a bound on runtime, we obviously prefer the algorithm that can solve the most problems within that bound. A similar metric is commonly used in SAT solver competitions.

The results show that using topological structure enumeration can significantly improve the solving time, as well as the number of solved instances. For *NPN4*, our fence-based algorithm is more than 19% faster than our baseline implementation. All algorithms find the solutions for all problem instances. For *FDSD6*, *Fence* is 2× faster than *SSV*. Again, there are no timeouts. For *PDSD6*, *Fence* is also 2× faster than *SSV* and we also have 2× fewer timeouts. The same observation can be made for the 8-input function sets. For *FDSD8*, *Fence* is again 2× faster than *SSV*. Finally, for *PDSD8*, *Fence* is 63.43% faster than *SSV*. Again, fence-based synthesis has fewer timeouts. In fact, the table shows that it dominates SSV with respect to the number of solved instances. In summary, we see that the gains from using topological constraints can be substantial.

G. Synthesis With Partial DAGs

Here, we compare synthesis based on partial DAGs to fence-based synthesis and conventional encodings. First, we apply partial DAG synthesis on the benchmarks described in Section IV-F. Table VI contains the results. Partial DAGs allow us to improve runtimes on the NPN4 and PDSD6 benchmarks. On NPN4, partial DAGs obtain a runtime reduction of 3× over both SSV and Fences. On PDSD6, the runtime reductions are 12× and 5.5×, respectively. Moreover, on the PDSD6 benchmark, they reduce the number of timeouts by 251 and 123 as compared to SSV and Fences, synthesizing all but 5 of the functions in under 3 min. Partial DAGs perform less well than SSV particularly on the FDSD8 and PDSD8 benchmarks. We conjecture that this is caused by the larger combinational complexity of the functions in those benchmarks. This forces partial DAG synthesis to try more topologies, thus slowing it down. However, we believe that our filtering methods can likely still be improved to further reduce the number of potential remedies.

In our next experiment, we compare SSV, fence-based, and partial DAG-based synthesis on a hard benchmark set. We sample 500 random 5-input functions, and try to synthesize optimum 2-input operator chains. Note that the majority of 5-input functions are hard, in that they require a large number of gates to implement [9]. In fact, it is true in general that most functions are random, and that random functions require exponentially many gates [39]. In this experiment, we see how many functions these different methods can synthesize, setting a timeout at 1 min. Fig. 6 shows the results. We see that synthesis based on partial DAGs is able to synthesize more than 3× as many functions in under 1 min of runtime. We conclude that both fences and partial DAGs can unlock significant runtime improvements and can both be used to solve more problem instances, although the domains on which they are best used may be different.

V. TOPOLOGY-BASED PARALLEL EXACT SYNTHESIS

In this section, we outline and evaluate a parallel exact synthesis architecture based on topology families. We do not

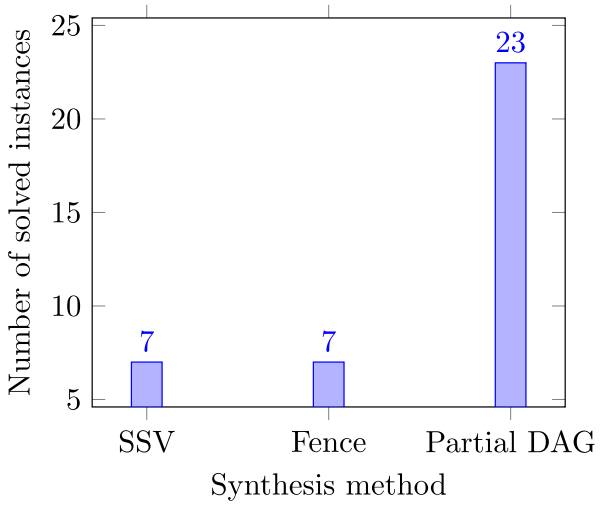

Fig. 6. Shows, for a set of 500 hard benchmarks, the number of successfully synthesized chains within the 1 min timeout.

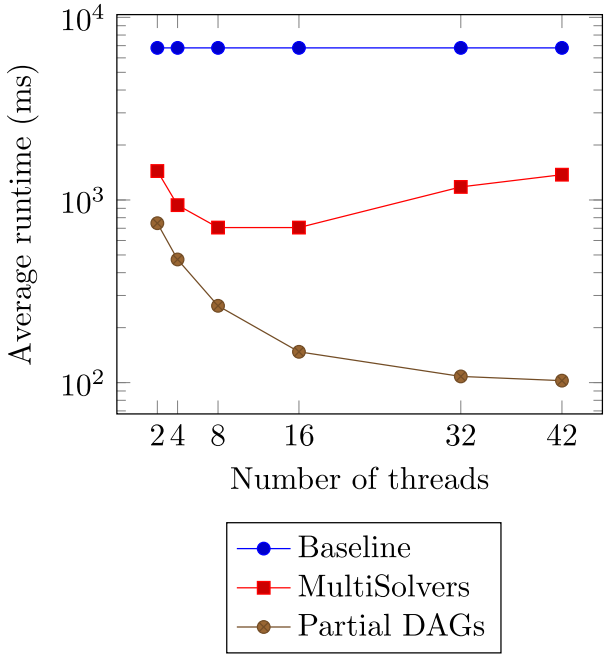

Fig. 7. Comparison between our domain-specific parallelism and a generic parallel SAT backend.

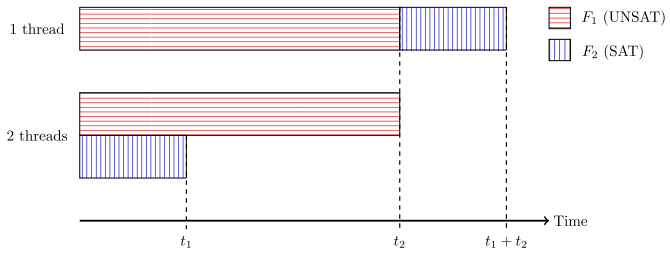

Fig. 8. Consider two topologies, F_1 and F_2, where F_2 can be used to synthesize a function, but F_1 cannot. Synthesizing sequentially, we must solve an UNSAT formula before an SAT one, which takes time $t_1 + t_2$. In a 2-threaded scenario, we can stop after $t_1 < [(t_1 + t_2)/2]$ time, leading to a super-linear speedup.

assume anything about the type of topology family. They may be fences, partial DAGs, or some other kind of topologies.

Suppose we are given a function f to synthesize. We can then produce a stream of topologies that may be used as a basis for f, as described in Section IV-A. In this scenario it will be useful to consider the stream as a queue Q. We do not know in advance which topology can implement f. Therefore, the single-threaded algorithms above sequentially pop topologies out of Q until they find one that applies. Now, suppose, we have n threads, all of which have access to Q. They can all pop topologies out of Q until one of them finds a topology that works. As soon as a solution is found by thread t it can signal the other threads to stop working. In fact, the situation is slightly more nuanced. To guarantee a minimum solution, threads t' that are looking for solutions with fewer gates than t should not be stopped. Alternatively, we may stage the generation of topologies, first generating all topologies with one gate, then those with two gates, and so on. Generating stages in sequence, we can stop as soon as the first thread in a stage finds a solution. This second approach was used in our experiments here. This algorithm is embarrassingly parallel, as there are no dependencies between threads, and there is no communication required except for the signal that a solution has been found.

1) Topology-Based Versus Generic Parallelism: The algorithm we describe above is one of many possible approaches to parallel SAT-based exact synthesis. Another is to use a generic parallel SAT solver to solve the CNF formulas generated by some encoding. However, we conjecture that such an approach is suboptimal, as such a solver is domain independent. To verify this hypothesis, we synthesize 2-input operator chains for a set of 1000 5-input functions, using two different parallel synthesis approaches. The first uses the SSV encoding, with a parallel SAT solver backend. We use Glucose-Syrup MultiSolvers, which won gold in the parallel track of the 2017 SAT competition [40], [41]. The second uses our proposed parallel architecture, with partial DAGs as topology families. Each thread is assigned its own single-threaded SAT solver. We use the bsat solver, taken from ABC [42]. Fig. 7 contains the results. It also shows, as a baseline, the single-threaded performance of the bsat solver using the SSV encoding.

The results show that the MultiSolvers and partial DAG implementations are up to 9.5 and 68× faster than the single-thread baseline, respectively. The partial DAG implementation is up to 7× faster than the best MultiSolvers configuration. Moreover, we see better scaling properties. The performance of partial DAG synthesis roughly doubles each time we double the number of threads. We do not see the same behavior using the MultiSolvers backend. In fact, its performance degrades after adding more than 16 threads. This is likely caused by increased thread contention as well a higher memory overhead as compared to our partial DAG implementation.

Interestingly, our implementation achieves a speedup of 68× as compared to the single-thread baseline, even though it uses at most 42 threads. In other words, it obtains a super-linear speedup. To see how this is possible, consider Fig. 8.

2) Majority-7 Decomposition: Two major applications of exact synthesis are synthesis with novel logic primitives and finding new upper bounds for classes of circuits. Our second experiment in this section considers both of these objectives.

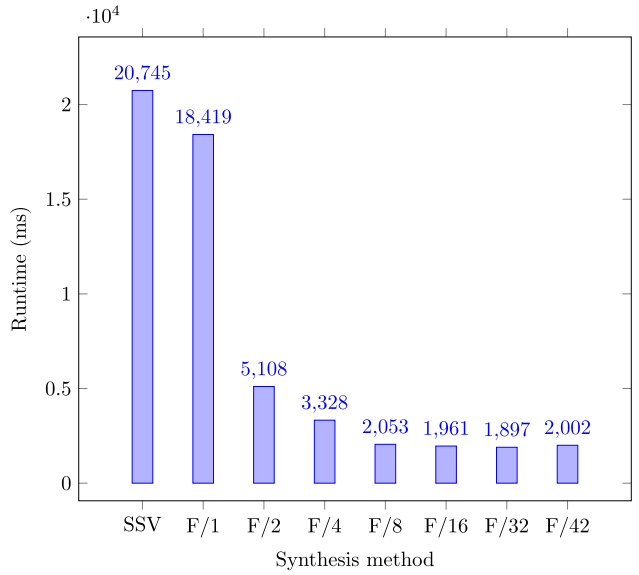

Fig. 9. Comparison of majority-7 decomposition between the best SSV encoding and a fence-based encoding with an increasing number of threads.

It concerns the decomposition of majority-n functions. Recall that the majority-n function is defined as

$$\langle x_1 \dots x_n \rangle = \left[x_1 + \dots + x_n > \frac{n-1}{2} \right] (n \text{ odd}).$$

One often wants to find a decomposition of majority-n functions into majority-3 operations, as this is an important task in majority-based logic synthesis. This has applications in both classical logic synthesis as well as synthesis for emerging technologies [43]. Moreover, upper bounds for small circuits can help us find better theoretical upper bounds for larger ones [20]. Therefore, in this experiment we decompose the majority-7 function into an optimum network of majority-3 operators. We use the same parallel exact synthesis architecture as before, but this time using fences as the topology families. To show the impact of parallelism we attempt this decomposition with increasing numbers of threads. We compare against a conventional synthesis method that is based on an extension of the SSV encoding. The results can be found in Fig. 9. In this figure, F/x refers to fence-based synthesis with x threads. The conventional approach requires 20 745 ms. The single-threaded fence-based approach is 11% faster, showing again the impact that topology-based synthesis can have even in the single-threaded case. With two threads, the fence-based synthesis is about 4× faster. This is another example topology-based multithreading unlocking super-linear speedups. Moreover, as we double the number of threads, synthesis time is cut approximately in half until we reach 16 gates. As we increase to 32 threads, runtime still decrease, but not as significantly. Finally, when we reach 42 threads, we slightly degrade performance. We conjecture that the added cost of creating more threads outweighs the additional throughput they provide. The best runtime, 1897 ms, is achieved by 32 threads. Thus, we achieve a runtime reduction of more than 10×.

VI. Discussion

In this paper, we take a new look at the difficult problem of SAT-based exact synthesis. We find that there are significant differences between encodings (and symmetry breaks) which can affect runtime by up to 3.5× (between encodings) and 72× (between symmetry breaking configurations). This is not yet the final word on encodings comparisons. Techniques, such as lazy addition of constraints are known to improve runtimes but are outside the scope of this paper.

We introduce an SAT-based exact synthesis method based on topological structure enumeration. Since the number of topological structures grows very quickly as the number of gates increases, we collect sets of structures in *topology families*. This paper introduces a theory of Boolean fences and partial DAGs and shows how they are used to constrain the CNF encodings. We find that the use of topology families can reduce synthesis runtime by up to 2× and improves the number of successfully synthesized problems by up to 51×. Moreover, topology-based synthesis is flexible and can be adapted to various encodings. Thus, we can create different topology-based synthesis flows for different domains.

Finally, we show how topology families can be used to transform the exact synthesis problem into a parallel one. We show that topology-based parallelism is up to 7× and 68× faster than a generic parallel SAT solver and a single-threaded algorithm, respectively. These improvements have direct impact on a variety of logic optimization algorithms that use exact synthesis, such as logic rewriting, technology mapping, and synthesis for emerging technologies [14]–[17]. There may be other ways to exploit parallelism. For example, one can imagine an approach which uses different encodings in parallel, thus creating a virtual best encoding.

Acknowledgment

The authors would like to thank A. Mokhov for his insights regarding CNF cardinality constraints. They are also grateful for the guidance provided by L. Amarù, J. Luo, and J. Olson from Synopsys Inc.

References

[1] J. Miao, A. Gerstlauer, and M. Orshansky, "Approximate logic synthesis under general error magnitude and frequency constraints," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, San Jose, CA, USA, 2013, pp. 779–786.

[2] W. V. Quine, "The problem of simplifying truth functions," *Amer. Math. Monthly*, vol. 59, no. 8, pp. 521–531, 1952.

[3] E. J. McCluskey, "Minimization of Boolean functions," *Bell Syst. Techn. J.*, vol. 35, no. 6, pp. 1417–1444, 1956.

[4] T. Sasao, "EXMIN2: A simplification algorithm for exclusive-OR-sum-of products expressions for multiple-valued-input two-valued-output functions," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 12, no. 5, pp. 621–632, May 1993.

[5] R. Ashenhurst, "The decomposition of switching functions," in *Proc. Int. Symp. Theory Switching*, 1957, pp. 74–116.

[6] A. Curtis, *New Approach to the Design of Switching Circuits*. Princeton, NJ, USA: Van Nostrand, 1962.

[7] E. S. Davidson, "An algorithm for NAND decomposition under network constraints," *IEEE Trans. Comput.*, vol. C-18, no. 12, pp. 1098–1109, Dec. 1969.

[8] J. P. Roth and R. M. Karp, "Minimization over Boolean graphs," *IBM J. Res. Develop.*, vol. 6, no. 2, pp. 227–238, 1962.

[9] D. E. Knuth, *The Art of Computer Programming*, vol. 4A. Upper Saddle River, NJ, USA: Addison-Wesley, 2011.

[10] L. Amarú *et al.*, "Enabling exact delay synthesis," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Irvine, CA, USA, 2017, pp. 352–359.

[11] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York, NY, USA: McGraw-Hill, 1994.

[12] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proc. IEEE*, vol. 78, no. 2, pp. 264–300, Feb. 1990.
[13] N. Song and M. A. Perkowski, "EXORCISM-MV-2: Minimization of exclusive sum of products expressions for multiple-valued input incompletely specified functions," in *Proc. ISMVL*, 1993, pp. 132–137.
[14] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Improvements to technology mapping for LUT-based FPGAs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 2, pp. 240–253, Feb. 2007.
[15] W. Haaswijk, M. Soeken, L. Amarú, P.-E. Gaillardon, and G. De Micheli, "A novel basis for logic rewriting," in *Proc. ASPDAC*, 2017, pp. 151–156.
[16] M. Soeken, L. Amarù, P.-E. Gaillardon, and G. De Micheli, "Exact synthesis of majority-inverter graphs and its applications," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 36, no. 11, pp. 1842–1855, Nov. 2017.
[17] M. Soeken, G. De Micheli, and A. Mishchenko, "Busy man's synthesis: Combinational delay optimization with SAT," in *Proc. Design Autom. Test Europe*, 2017, pp. 830–835.
[18] M. Soeken *et al.*, "Practical exact synthesis," in *Proc. Design Autom. Test Europe Conf. Exhibit. (DATE)*, Dresden, Germany, Mar. 2018, pp. 309–314.
[19] K. Stoffelen, *Optimizing S-Box Implementations for Several Criteria Using SAT Solvers* (LNCS 9783). Heidelberg, Germany: Springer, 2016, pp. 140–160.
[20] A. S. Kulikov, "Improving circuit size upper bounds using sat-solvers," in *Proc. Design Autom. Test Eur. Conf. Exhibit. (DATE)*, Mar. 2018, pp. 305–308.
[21] T. Sasao, *Switching Theory for Logic Synthesis*. New York, NY, USA: Springer, 1999.
[22] A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability*. Amsterdam, The Netherlands: IOS Press, 2009.
[23] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, *Counterexample-Guided Abstraction Refinement*. Heidelberg, Germany: Springer, 2000, pp. 154–169.
[24] Y. Hamadi, "ManySAT: A parallel SAT solver," *J. Satisfiability Boolean Model. Comput.*, vol. 6, no. 5, pp. 245–262, 2009.
[25] M. J. H. Heule, O. Kullmann, S. Wieringa, and A. Biere, *Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads*, K. Eder, J. Lourenço, and O. Shehory, Eds. Berlin, Germany: Springer, 2012.
[26] G. Katsirelos, A. Sabharwal, H. Samulowitz, and L. Simon, "Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers," in *Proc. 27th AAAI Conf. Artif. Intell.*, 2013, pp. 481–488.
[27] W. Haaswijk, E. Testa, M. Soeken, and G. De Micheli, "Classifying functions with exact synthesis," in *Proc. ISMVL*, 2017, pp. 272–277.
[28] N. Eén, "Practical SAT—A tutorial on applied satisfiability solving," in *Proc. FMCAD*, 2007.
[29] A. Kojevnikov, A. S. Kulikov, and G. Yaroslavtsev, "Finding efficient circuits using SAT-solvers," in *Proc. Theory Appl. Satisfiability Test.*, 2009, pp. 32–44.
[30] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Reading, MA, USA: Addison-Wesley, 2015.
[31] M. D. Riedel, "Cyclic combinational circuits," Ph.D. dissertation, Elect. Eng., California Inst. Technol., Pasadena, CA, USA, 2004.
[32] C. Sinz, "Towards an optimal CNF encoding of Boolean cardinality constraints," in *Proc. Principles Pract. Constraint Program. (CP)*, 2005, pp. 827–831.
[33] L. Amarù, P.-E. Gaillardon, A. Chattopadhyay, and G. De Micheli, "A sound and complete axiomatization of majority-*n* logic," *IEEE Trans. Comput.*, vol. 65, no. 9, pp. 2889–2895, Sep. 2016.
[34] B. D. McKay and A. Piperno, "Practical graph isomorphism, II," *J. Symbolic Comput.*, vol. 60, pp. 94–112, Jan. 2014.
[35] E. M. Luks, "Isomorphism of graphs of bounded valence can be tested in polynomial time," *J. Comput. Syst. Sci.*, vol. 25, no. 1, pp. 42–65, 1982.
[36] W. J. Haaswijk, A. Mishchenko, M. Soeken, and G. De Micheli, "SAT based exact synthesis using DAG topology families," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, San Francisco, CA, USA, 2018, pp. 1–6.
[37] Z. Huang, L. Wang, Y. Nasikovskiy, and A. Mishchenko, "Fast Boolean matching based on NPN classification," in *Proc. Int. Conf. Field Program. Technol.*, 2013, pp. 310–313.
[38] A. Mishchenko, "An approach to disjoint-support decomposition of logic functions," Elect. Eng. Comput. Sci., Portland State Univ., Portland, OR, USA, Rep., 2001.
[39] J. Riordan and C. E. Shannon, "The number of two-terminal series-parallel networks," *J. Math. Phys.*, vol. 1, no. 4, pp. 83–93, 1942.
[40] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *Proc. 21st Int. Joint Conf. Artif. Intell. (IJCAI)*, 2009, pp. 399–404.
[41] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*. Heidelberg, Germany: Springer, 2004, pp. 502–518.
[42] R. K. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Proc. Comput.-Aided Verification*, 2010, pp. 24–40.
[43] M. Soeken, E. Testa, A. Mishchenko, and G. De Micheli, "Pairs of majority-decomposing functions," *Inf. Process. Lett.*, vol. 139, pp. 35–38, Nov. 2018.

Winston Haaswijk (S'14) received the bachelor's degree in computer science from the University of Amsterdam, Amsterdam, The Netherlands, and the M.Phil. degree in computer science from the University of Cambridge, Cambridge, U.K. He is currently pursuing the Ph.D. degree with the Integrated Systems Laboratory, EPFL, Lausanne, Switzerland.

His current research interests include Boolean satisfiability, exploring novel logic primitives, SAT-based synthesis methods, machine learning in general, and applications of machine learning to EDA in particular. He maintains percy, which is a C++ header-only SAT-based exact synthesis library, and one of the EPFL logic synthesis libraries.

Mathias Soeken (S'09–M'13) received the Ph.D. degree in computer science and engineering from the University of Bremen, Bremen, Germany, in 2013.

He is a Scientist with the École Polytechnique Fédéderale de Lausanne (EPFL), Lausanne, Switzerland. He is investigating constraint-based techniques in logic synthesis and industrial-strength design automation for quantum computing. He is actively maintaining the logic synthesis frameworks CirKit and RevKit. His current research interests include the many aspects of logic synthesis and formal verification.

Dr. Soeken was a recipient of the scholarship from the German Academic Scholarship Foundation. He has been serving as a TPC member for several conferences, including DAC, DATE, and ICCAD and is a Reviewer for *Mathematical Reviews* as well as for several other journals.

Alan Mishchenko (M'98–SM'14) received the M.S. degree from the Moscow Institute of Physics and Technology, Moscow, Russia, in 1993 and the Ph.D. degree from the Glushkov Institute of Cybernetics, Kiev, Ukraine, in 1997.

In 2002, he joined the EECS Department, University of California at Berkeley, Berkeley, CA, USA, where he is currently a Full Researcher. His current research interests include computationally efficient logic synthesis and formal verification.

Giovanni De Micheli (S'82–M'83–SM'89–F'94) received the Nuclear Engineering degree from the Politecnico di Milano, Milan, Italy, in 1979 and the M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California at Berkeley, Berkeley, CA, USA, in 1980 and 1983, respectively.

He was a Professor of electrical engineering with Stanford University, Stanford, CA, USA. He is a Professor and the Director of the Institute of Electrical Engineering, École Polytechnique Fédéderale de Lausanne, Lausanne, Switzerland.

Prof. De Micheli was a recipient of the 2016 IEEE/CS Harry Goode Award for Seminal Contributions to Design and Design Tools of Networks on Chips, the 2016 EDAA Lifetime Achievement Award, and other awards. He is a fellow of ACM and a member of the Academia Europaea.