Progressive Generation of Canonical Sums of Products Using a SAT Solver

Ana Petkovska[1] ana.petkovska@epfl.ch
Alan Mishchenko[2] alanmi@berkeley.edu
David Novo[3] david.novo@lirmm.fr
Muhsen Owaida[4] mohsen.ewaida@inf.ethz.ch
Paolo Ienne[1] paolo.ienne@epfl.ch

[1]Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences, Lausanne, Switzerland
[2]University of California, Berkeley, Department of EECS, Berkeley, USA
[3]Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier, Montpellier, France
[4]Eidgenössische Technische Hochschule Zürich (ETHZ), Zürich, Switzerland

ABSTRACT

We present an algorithm that progressively generates canonical *Sums Of Products (SOPs)* for completely- and incompletely-specified Boolean functions using a satisfiability solver. The progressive generation allows for real time monitoring and early termination, as well as for generation of partial SOPs for specific incremental applications. On the other hand, canonicity brings independence of the original representation and typically yields more regular SOPs that factor better. Also, canonicity is key in applications as constraint solving and random assignment generation, which traditionally rely on BDD-based methods. However, in contrast with BDDs, our algorithm can relax canonicity to improve speed and scalability. On average, our method improves the quality of results up to 10%, in terms of SOP size, compared to a state-of-the-art BDD-based method. Experiments with global circuit restructuring based on SAT-based SOPs show that area-delay product can be improved up to 36%, compared to using BDD-based SOPs.

1. INTRODUCTION

Minimization of the two-level *Sum Of Products (SOP)* representation is well studied due to the wide use of SOPs. In the past, research in SOPs was motivated by mapping into *Programmable Logic Arrays (PLAs)*; now SOPs are supported in many tools for logic optimization and are used for multi-level logic synthesis [3,21], delay optimization [16], test generation [7] and in other areas.

Contrary to the popular belief, research in SOP minimization and its applications is not outdated. As an example, a recent work uses SOPs for delay optimization in technology independent synthesis and technology mapping [16]. In this work, improved quality is achieved by enumerating different SOPs of the local functions of the nodes, factoring them, and finding circuit structures balanced for delay.

Another important application of SOP minimization, targeted in this paper, is *global circuit restructuring.* If a multi-level circuit structure for a Boolean function is not available, or if the circuit structure is redundant, a new circuit structure with desirable properties, such as low area, short delay, good testability or improved implicativity (if the circuit represents constraints in a SAT solver) should be derived.

The best known (and widely used) method for global circuit restructuring is computing SOPs of the output functions in terms of inputs, factoring the multi-output SOPs and deriving a new circuit structure from the shared factored form. The main drawback of this method is the lack of scalability of the *Binary Decision Diagram (BDD)* construction, which is required to compute the SOPs of the output functions. To our knowledge, this method is used in most of the industrial tools, and therefore scalability improvements of this method are highly desirable.

Additional drawback is that BDDs are incompatible with incremental applications since they require building a BDD for the complete circuit before converting it to SOP. Furthermore, for some circuits, the BDD construction suffers from the BDD memory explosion problem—the BDD size is exponential in the number of input variables—and thus, using BDDs is often impractical.

Alternatively, recent progress in the performance of *Boolean satisfiability (SAT) solvers* enabled using SAT in various domains of logic synthesis and verification despite their worst-case exponential runtime. Thus, it has become a trend to replace BDD-based methods with SAT-based ones. For example, this was done for model checking [14], functional dependency [8], functional decomposition [11, 12] and logic don't-care-based optimization [17]. Existing methods for SOP generation using SAT solvers are based on enumeration of satisfying assignments [18]. On the other hand, Sapra et al. [23] proposed SAT-based algorithm for part of ESPRESSO's procedures for SOP minimization. But, since they largely follow the traditional ESPRESSO style of SOP minimization, they operate on existing SOPs and do not consider generating a new SOP from a multi-level representation of the Boolean function. To the best of our knowledge, there is still no complete SAT-based method for SOP generation similar to the *Irredundant Sum-of-Product (ISOP)* algorithm for incompletely specified functions using BDDs [15].

Accordingly, the main contribution of this paper is to propose a new engine for SOP generation and minimization that is completely based on SAT solvers. Our method generates the SOP progressively, building it cube by cube. We guarantee that the generated SOPs are *irredundant*, meaning that no literal and no cube can be deleted without changing the function. As we show in the result section, our algorithm generates SOPs with the size similar to that of the BDD-based method [15]. Interestingly, for some circuits, we generate smaller SOPs (up to 10%), which is useful in practical applications. For example, when a multi-level description of the circuit is built using an SOP produced by the proposed SAT-based method, the area-delay product of the resulting circuit, assuming unit-area and unit-delay model, decreases up to 36%, compared to using BDDs.

Two main features characterize our SAT-based SOP generation and make it desirable in various domains.

First, we generate an SOP *progressively*, unlike BDD-based methods that attempt to construct a complete SOP at once. The progressive computation allows generation of *a partial SOP* for circuits whose complete SOP cannot be computed given the resource limits. The partial SOPs can be exploited by other applications that do not require the complete circuit functionality, but work with partially defined functions [4,25]. Moreover, for circuits with large SOPs, the progressive generation allows us to predict whether it is feasible to build an SOP for a circuit, and to check if the SOP size is within the limits of the methods that are going to use it. For this, at any moment, we can retrieve the number of outputs for which the SOP is already computed, as well as the finished SOP portion of the currently processed output. We can also easily compute an estimate or a lower-bound of the percentage of covered minterms, considering uniform distribution of minterms in the space or considering the size of the truth table, respectively. In contrast, the termination time and the quality of results of the BDD-based methods are unpredictable since the complete BDD has to be built before converting it to SOP.

Second, counter-intuitive as it may sound, we show that the SAT-based computation can generate *canonical* SOPs. To this end, we combine (1) an algorithm that, under a given variable order, generates consecutive SAT assignments in lexicographic order [20], considering each assignment as integer value, and (2) a deterministic algorithm that expands the received assignments into cubes. For a given function and a variable order, the assignments (i.e., the minterms) are always generated in the same order, and each assignment always results in the same cube. Thus, the resulting SOP is canonical—it is unique and independent of the input implementation of the function. The canonical nature of the resulting SOPs can be useful in those domains where previously only BDDs could be used. For example, constraint solving [26] and random assignment generation [19] can benefit from the canonicity if we iterate repeated generation of random valuation of inputs and get the closest SAT assignment, as it is done in the proposed canonical SOP generation method. Also, the canonicity brings regularity in the SOPs, and thus the results after using algorithms for factoring [21] are expected to better.

In the rest of the paper, we focus on completely-specified functions, but it should be noted that the given SAT-based formulation works for incompletely-specified functions without any changes. Indeed, after extracting the first cube and blocking it in the on-set of the function, the rest of the computation is performed for the incompletely-specified functions, even if the initial function was completely specified. A more interesting extension is to enumerate cubes that are shared across multiple outputs. Implementing this extension and exploring the resulting improvements in quality is left for future work.

The rest of the paper is organized as follows. Section 2 gives background on Boolean functions, the SOP representation and the satisfiability problem. Next, we describe our algorithm for SAT-based progressive generation of irredundant SOPs in Section 3. Section 4 gives our experimental setup and discusses the experimental results. Section 5 gives additional information on the related work. Finally, we conclude and present ideas for future work in Section 6.

2. BACKGROUND INFORMATION

In this section, we define the terminology associated with Boolean functions and the SOP representation, as well as the satisfiability problem.

2.1 Boolean Functions

For a variable v, a *positive literal* represents the variable v, while the *negative literal* represents its negation $\bar{v}$. A *cube*, or a product, c, is a Boolean product (AND, $\cdot$) of literals, $c = l_1 \cdot \dots \cdot l_k$. If a variable is not represented by a negative or a positive literal in a cube, then it is represented by a *don't-care* $(-)$, meaning that it can take both values 0 and 1. A cube with i don't-cares, covers 2^i minterms. A *minterm* is the smallest cube in which every variable is represented by either a negative or a positive literal.

Let $f(X) : B^n \to \{0, 1, -\}$, $B \in \{0, 1\}$, be an *incompletely specified Boolean function* of n variables $X = \{x_1, \dots, x_n\}$. The terms function and circuit are used interchangeably in this paper. The *support set* of f is the subset of variables that determine the output value of the function f. The set of minterms for which f evaluates to 1 defines the *on-set* of f. Similarly, the minterms for which f evaluates to 0 and don't-care define the *off-set* and the *don't-care-set*, respectively. In a multi-output function $F = \{f_1, \dots f_m\}$, each output f_i, $1 \le i \le m$, has its own support set, on-set, off-set and don't-care-set associated with it.

For simplicity, we define the following terms for single-output functions, although our algorithm can handle multi-output functions. Any Boolean function can be represented as a two-level *sum of products (SOP)*, which is a Boolean sum (OR, +) of cubes, $S = c_1 + \cdots + c_k$. Assume that a Boolean function f is represented as an SOP. A cube is *prime*, if no literal can be removed from the cube without changing the value that the cube implies for f. A cube that is not prime, can be *expanded* by substituting at least one literal with a don't-care. The SOP is *irredundant* if each cube is prime and no cube can be deleted without changing the function.

A *canonical representation* is a unique representation for a function under certain conditions. For example, given a Boolean function and a fixed input variable order, a canonical SOP is an SOP independent of the original representation of the function given to the SAT solver. In a similar way, BDDs can be used to generate a canonical SOP that only depends on an input variable order [15].

2.2 Boolean Satisfiability

A disjunction (OR, +) of literals forms a *clause*, $t = l_1 + \cdots + l_k$. A *propositional formula* is a logic expression defined over variables that take values in the set {0, 1}. To solve a SAT problem, a propositional formula is converted into its *Conjunctive Normal Form (CNF)* as a conjunction (AND, $\cdot$) of clauses, $F = t_1 \cdot \dots \cdot t_k$. Algorithms such as the Tseitin transformation [24] convert a Boolean function into a set of CNF clauses.

A *satisfiability (SAT) problem* is a decision problem that takes a propositional formula in CNF form and returns that the formula is *satisfiable* (*SAT*) if there is an assignment of the variables from the formula for which the CNF evaluates to 1. Otherwise, the propositional formula is *unsatisfiable* (*UNSAT*). A program that solves SAT problems is called a *SAT solver*. SAT solvers provide a *satisfying assignment* when the problem is satisfiable.

Modern SAT solvers can determine the satisfiability of a problem under given assumptions. *Assumptions* are propositions that are given as input to the SAT solver for a specific single invocation of the SAT solver and have to be satisfied for the problem to be SAT.

EXAMPLE 1. *For the function* $f(x_1, x_2, x_3) = (x_1 + x_2)\bar{x}_3$*, which is satisfiable for the following assignments of the inputs* $\{(0,1,0),(1,0,0),(1,1,0)\}$*, a SAT solver without assumptions can return any of the given assignments. But, if we give as input to the SAT solver the assumption* $x_1 = 1$*, then it returns either* $(1,0,0)$ *or* $(1,1,0)$*, because those two assignments satisfy the given assumption.*

A *lexicographic satisfiability (LEXSAT) problem* is a SAT problem that takes a propositional formula in CNF form and, given a variable order, returns a satisfying variable assignment whose integer value under the given variable order is minimum (maximum) among all satisfiable assignments. If the formula has no satisfiable assignments, LEXSAT proves it unsatisfiable. The LEXSAT problem was first mentioned by Knuth [9]. For our work, we use an efficient algorithm and methods for generating consecutive SAT assignments in lexicographic order [20].

EXAMPLE 2. *For the function* $f(x_1, x_2, x_3)$ *from Example 1, LEXSAT returns either the lexicographically smallest assignment* $(0,1,0)$ *or the lexicographically largest assignment* $(1,1,0)$*, depending on the user preference.*

3. SAT-BASED SOP GENERATION

In this section, we describe our SAT-based algorithm that progressively generates an irredundant SOP for a single-output function. For multi-output circuits, each output is treated separately. In this paper, we focus on completely-specified functions, but the algorithm can be easily used for incompletely-specified functions by providing both the on-set and off-set as input to the algorithm. In the case of a completely specified function one of them is derived by complementing the other.

The presented algorithm iteratively generates minterms, expands them into prime cubes, and adds these cubes to the SOP. The SAT-based heuristics for minterm generation and cube expansion are described in Section 3.1 and Section 3.2, respectively. Finally, to guarantee that the resulting SOP is irredundant, it is post-processed to remove redundant cubes, as described in Section 3.3. Additionally, Section 3.4 describes several techniques that reduce the runtime. The procedures described in the following subsections assume that we are generating the on-set SOP. The same procedures are used to generate the off-set SOP.

The algorithm can be implemented with one SAT solver parameterized to store both on-set and off-set. Alternatively, it can use two solvers, one for on-set and one for off-set. In our implementation of the algorithm, we use four different SAT solvers: for both on-set and off-set, one is used to generate satisfying assignments, the other to expand assignments to cubes. By employing four solvers, we ensure that assignment generation and expansion do not interact with each other during the SOP computation.

3.1 Generation of Minterms

In order to generate minterms for a function f by using a SAT solver, we initialize a SAT solver with the CNF of f. Then, to discard the trivial case when the function has a constant on-set, we solve the SAT problem by asserting that $f = 1$. If the problem is UNSAT, then f is a constant, and we return an SOP with one constant cube.

Generation of non-canonical SOP. Otherwise, if the problem is SAT, an assignment for the inputs is returned for which the function evaluates to 1. From the assignment, we can generate a minterm for the function f in which the variables assigned to 0 and 1 are represented with the negative and positive literal, respectively. For example, for a function $f(x, y, z)$, the assignment $(1,1,0)$ implies the minterm $xy\bar{z}$. Once a minterm is obtained, we expand it into a cube using the heuristic procedure from Section 3.2. Next, we add the cube with its literals complemented to the SAT solver as a *blocking clause*, which is an additional clause that blocks known solutions of the SAT problem. This allows to generate the next minterm that is not covered by any of the previously generated cubes. While the problem is SAT, we iteratively obtain a minterm, expand it to a cube, and add the cube to both the SAT solver and the SOP. The unsatisfiability of the problem indicates that the generated SOP is complete and covers all on-set minterms.

Generation of canonical SOP. However, generating minterms from satisfying assignments received from a SAT solver does not guarantee canonicity, since SAT solvers return minterms in a non-deterministic order that depends on the design of the solver and the CNF generated for the function. Thus, to ensure canonicity, we iteratively use a binary search-based LEXSAT algorithm, called `BINARY` [20], that generates minterms in a lexicographic order that is unique for a given variable order. The algorithm `BINARY` receives as input a potential assignment, which is the lexicographically smallest assignment that might be satisfiable, that is either the last generated minterm or, initially, an assignment with all 0s. Then, `BINARY` tries to verify and fix the assignment of each variable defined with the potential assignment starting from the leftmost variables and moving to right. We also use the proposed methods for runtime improvement [20]: skip verifying the leading 1s, correcting the initial potential assignment, and profiling the success of the first SAT call. Similarly to the non-canonical SOPs, once we obtain a minterm, we expand it into a cube and add it to the SAT solver as a blocking clause.

EXAMPLE 3. *For example, assume that for the function* $f(x_1, \ldots, x_8)$*, the last generated minterm* $(1,1,0,0,0,0,0,1)$ *is received as an initial potential assignment. Since this minterm is covered by the last cube, this assignment is not satisfiable, so we can increase its value for 1 to get the smallest assignment that might be satisfiable* $(1,1,0,0,0,0,1,0)$*. Next, we can skip verifying the assignments* $x_1 = 1$ *and* $x_2 = 1$*, because the next lexicographically smallest assignment has to start with the same leading 1s. Thus, we should* *only check the assignments for x_i, for $3 \leq i \leq 8$. Due to using binary search, with the first SAT call we assume half of the unverified assignments, and we give to the on-set SAT solver the assumptions $(x_1, \ldots, x_5) = (1, 1, 0, 0, 0)$. Assume that the problem was satisfiable and the SAT solver returned the assignment $(1, 1, 0, 0, 0, 0, 1, 1)$. This assignment proves that an on-set minterm with the assumed values exists, but moreover we can learn that the assignments from the potential minterm $x_6 = 0$ and $x_7 = 1$ are correct. Next, to check if the assignment for the last input x_8 can be set to 0, we call the SAT solver with the assumptions $(x_1, \ldots, x_8) = (1, 1, 0, 0, 0, 0, 1, 0)$. If it returns SAT, we return the potential assignment as a minterm. Otherwise, we flip x_8 to 1 before returning it.*

3.2 Expansion of Minterms to Cubes

In this subsection, we describe our SAT-based procedure that receives a minterm and transforms it into a prime cube by iteratively removing literals (i.e., substituting them with don't-cares). For the on-set SOP, a literal can be removed, if after its removal all minterms covered by the cube belong to the on-set.

Greedy deterministic cube expansion. The heuristic removes literals in two rounds. First, we remove literals greedily, in the given order, after ensuring that additional on-set minterms are covered by expanding each literal.

EXAMPLE 4. *Assume that for the function on Figure 1, the cube c_1 was computed and added to the SAT solver, and as a second minterm $xyzt$ is generated, which can be extended by removing one of the literals x, y or t. If we remove x, we will obtain the cube c_4 that covers only one additional minterm with respect to the existing cube c_1, but if we remove y or t, we will obtain c_2 or c_5, respectively, each of which covers two additional minterms with respect to c_1.*

For Example 4, our expansion procedure skips the opportunity to remove the literal x, and tries to expand other literals if possible. To check if by removing a literal l_i, the expanded cube covers more than one new minterm, we flip l_i and provide it, along with the remaining literals of the cube, as assumptions to an on-set SAT solver that contains the already generated cubes. If the problem is SAT, then we consider this literal for removal since by removing it we cover more than one minterm. Otherwise, we skip removing it temporarily. Once we have found a literal that should be considered for removal, we perform the following *final SAT check* to ensure that it can be removed. First, we assume that the literal is removed from the cube. Next, we run the off-set SAT solver with assumptions for all remaining literals of the cube. If the problem is UNSAT, then no minterm covered by the cube belongs to the off-set, so we can extend the cube by removing this literal. On the other hand, if the problem is SAT, we cannot extend the cube, since the SAT solver found a minterm that belongs to the off-set and is covered by the extended cube.

Expansion to prime cubes. In the first round, we might skip some opportunities for expansion, but in the second round, for each remaining literal, we execute the final SAT check described above. This guarantees that no literal can be further removed, which means that the cube is prime. Since, we always try to remove the literals in the same order, this method generates a canonical SOP.

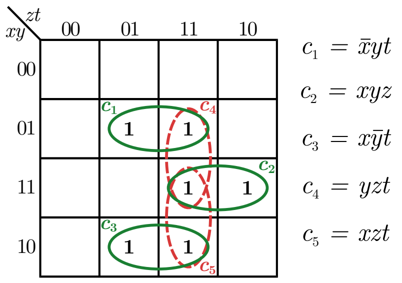

Figure 1: A Karnaugh map for the Boolean function $f(x, y, z, t) = \bar{x}yt + xyz + x\bar{y}t$ with its prime cubes c_i, where $1 \leq i \leq 5$. The cubes c_1, c_2 and c_3 are essential and they compose the minimum SOP of f.

Fast non-canonical expansion. If generating a canonical SOP is not required, we can substitute the first round of expansion with a faster method to improve runtime: If in an off-set SAT solver we assume the values from the received minterm, which is generated from the on-set, the problem is UNSAT and the SAT solver returns the set of literals used to prove unsatisfiability (procedure "analyse_final" in MiniSAT [6]). Thus, we can remove literals that are not returned by the SAT solver. However, the set of remaining literals is not minimal, and thus we run additionally the second round of removal with the final SAT check.

3.3 Removing Redundant Cubes

The cubes expanded with the methods from Section 3.2 are prime by construction. However, by progressively adding cubes to the SAT solver, as described in Section 3.1, we ensure that each cube is irredundant with respect to the preceding cubes, but not with respect to the whole set.

EXAMPLE 5. *For the function f from Figure 1, assume that the cubes c_1, c_5, c_2 and c_3 are generated in the given order. The cube c_5 is irredundant with respect to c_1, since it additionally covers the minterms $xyzt$ and $x\bar{y}zt$, but it is contained in the union of c_2 and c_3.*

In order to produce an irredundant SOP, after generating all cubes, we iterate through the cubes to detect and remove redundant ones. First, we initialize a new SAT solver with clauses for all generated cubes and we assume that all cubes are required. Then, by using the assumption mechanism, for each cube c_i, we check if there is an assignment for which c_i evaluates to 1 while all the other irredundant cubes evaluate to 0. If the problem is SAT, the cube is irredundant and the SAT solver returns an assignment that corresponds to a minterm which is covered only by c_i. Otherwise, if the problem is UNSAT, then the cube is redundant, and thus it is removed from the SOP and is excluded when checking the redundancy of the following cubes. Since we always try to remove cubes in the order in which they were generated, this method is deterministic and maintains canonicity when canonical SOPs are generated.

EXAMPLE 6. *Considering the cubes from Example 5, to check whether c_3 is redundant, we set $c_3 = 1$ by assuming the values $x = 1$, $y = 0$ and $t = 1$. For the assumed values, the other cubes evaluate to $c_1 = 0$, $c_2 = 0$ and $c_5 = z$. Setting $z = 0$ leads to $c_5 = 0$. Thus, the problem is SAT and c_3 is irredundant. The returned assignment $(x, y, z, t) = (1, 0, 0, 1)$ defines a minterm $x\bar{y}\bar{z}t$ that is covered only by c_3.*

3.4 Improving the Runtime

In this subsection, we present four techniques that improve the runtime of the algorithm by allowing early termination and by treating some special cases.

Simultaneous on-set and off-set generation. Often, the SOP of the on-set and off-set differ in size. For example, a two-input function implementing an AND gate, $f(x, y) = xy$, has on-set SOP, $f = S_{\text{on}} = xy$ with size 1, and off-set SOP, $\bar{f} = S_{\text{off}} = \bar{x} + \bar{y}$ with size 2. Thus, we propose to generate on-set and off-set cubes simultaneously, one cube at a time from each set. This way, if one of them is much smaller than the other, we can avoid the situation when the large set of cubes has to be first generated, before a smaller one is discovered.

Prioritizing outputs with large SOPs. Before generating SOPs for each output, we propose to sort outputs by size of their input supports. The outputs with larger supports are processed first since it is more likely that SOP generation for these outputs will exceed resource limits, so we can determine if we should terminate the computation earlier.

Isomorphic circuits. We implemented a method to decrease the runtime by detecting isomorphic outputs. For this, first, we divide the outputs into isomorphic classes. Two outputs are *isomorphic* and belong to the same class, if they implement an identical function using different inputs. Then, for each class, we generate an SOP only for one output, and duplicate it for the others. In Section 4.2, we show that this allows effective generation of an SOP only for 39.7% of the outputs. When running our SAT-based method on industrial benchmarks, which could not be quoted in the paper, we found that typically around 70% of the outputs are isomorphic, which means that this feature leads to at least 3x speedup.

CNF sharing. Generating a CNF for each output is time consuming. Thus, to benefit from the logic sharing among the outputs, we can optionally share one CNF, which corresponds to the complete circuit. For this, we generate the CNF of the circuit, and then, for each output, we initialize the SAT solver only with the part of the CNF for the corresponding output. Besides improving the runtime, as Table 1 shows, this option sometimes leads to better results.

4. EXPERIMENTAL RESULTS

In this section, we describe our experimental setup and compare the proposed SAT-based algorithm with a state-of-the-art BDD-based method.

4.1 Experimental Setup

We implemented the algorithm described in Section 3 as a new command *satclp* in ABC [2]. ABC is an open-source tool, designed for logic synthesis, technology mapping, and formal verification for logic circuits. ABC features an integrated SAT solver based on an early version of MiniSAT [6] that supports incremental SAT solving. Furthermore, ABC provides an implementation of BDD construction for a multi-level circuit (command *collapse*) and the BDD-based ISOP computation [15] (command *sop*). Finally, ABC allows us to analyze the area-delay results when the generated SOPs are used to build a new multi-level circuit representation. A multi-level network is generated using the *fx* command [21]. Then, it is converted into an and-inverter graph, which is

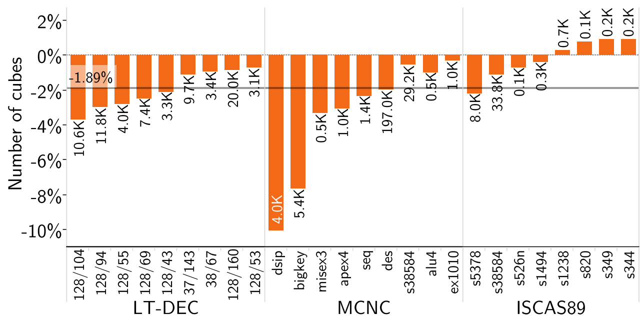

Figure 2: Size of the smallest SOPs generated by the SAT-based algorithm compared to the smallest SOP generated by the BDD-based method. Only the benchmarks for which the SOP size differs are shown. The gray line shows that, on average, the SAT-based method decreases the SOP size by 1.89%. Next to each bar we show the actual number of cubes for the SAT-based SOP.

an internal representation of ABC, and optimized with the *dc2* command. This shows how the difference in the SOPs affects the methods using them.

To evaluate our algorithm, we use the ISCAS'89 benchmarks, a set of large MCNC benchmarks and a set of 9 logic tables from the instruction decoder unit [1], which we denote with LT-DEC. The LT-DEC has been show as well-suited to demonstrate the factoring gains as circuit size increases [10]. The names of the LT-DEC benchmarks are given in the form "$[N_{\text{PI}}]/[N_{\text{PO}}]$", where N_{PI} is the number of primary inputs and N_{PO} is the number of primary outputs. For the main experiments, we discard benchmarks for which the SOP size exceeds the built-in resource limits of the command *sop* or *fx*, and thus, we use 30 (out of 32) and 14 (out of 20) benchmarks from the ISCAS'89 and MCNC set, respectively. With the discarded benchmarks, we demonstrate the generation of partial SOPs.

The reported runtime is the runtime returned by the command *time*. For the BDD-based method we call it before and after calling both *collapse* and *sop*. For our SAT-based algorithm, we call it before and after our command *satclp* that executes the algorithm. Thus, it reports the time for the complete algorithm, including detecting isomorphic outputs, generating CNF and initializing SAT solver instances, as well as the time for all SAT calls for cube generation, expansions and removing redundant cubes.

4.2 SAT- vs. BDD-based SOP Generation

To analyze the performance of the algorithm presented in Section 3, we run both the SAT-based algorithm and the BDD-based method available in ABC. In this section, we present the results of these experiments while analyzing the strengths and shortcomings of our approach.

Although the command *collapse* dynamically finds a good variable order for the BDD, changing the initial order of the primary inputs results in a different BDD structure, which leads to a different SOP. Thus, to obtain a good SOP, we generate five SOPs for the BDD-based method by using five different initial orders of the primary inputs. Similarly, our SAT-based algorithm generates different SOPs for different orders of the primary inputs, which define the order of re-

Table 1: **Number of benchmarks (out of the 53 used benchmarks) for which using a given combination of options for the SAT-based algorithm resulted in the smallest SOP in terms of number of cubes (column "#Cubes") or the best area-delay product (column "A·D"). When we select the best result, if we obtain same result with more than one combination of options, we give priority to the default one with least changes. The default options are using the initial order of the benchmark, no canonicity, no CNF sharing, and no reversing of the order.**

Order PI	Canonical	Shared CNF	Reverse	#Benchmarks #Cubes	A·D
Initial	No	No	No	15	15
			Yes	1	5
		Yes	No	-	3
			Yes	-	6
	Yes	No	No	23	12
			Yes	8	7
Fanouts	No	No	No	-	3
			Yes	1	1
	Yes	No	No	2	1
			Yes	3	-

Table 2: **Comparison of the number of combinational outputs, which are primary outputs and latch inputs, in the used benchmarks and the number of calls of the function for generating SOPs.**

	Comb. outputs	Generated SOPs	
LT-DEC	788	682	86.5%
MCNC	2779	1345	48.4%
ISCAS'98	5753	1675	29.1%
Total	9320	3702	39.7%

moving literals from the cubes. We either use the pre-defined order from the benchmark or order the inputs based on their number of fanouts, which currently only works for the combinational benchmarks. We can also, optionally, reverse the selected variable order. Moreover, we can enable or disable generation of canonical SOPs, and we can decide whether to generate one CNF for all outputs as described in Section 3.4. Thus, by trying different orders and changing the options, we generate 12 SOPs using the SAT-based method.

Generating multiple SOPs with each method results in SOPs that differ in size, where the SOP size is equal to the number of cubes that constitute the SOP. Figure 2 shows and compares the benchmarks for which the size of the smallest SOP generated by each method is different. We can see that the SAT-based algorithm decreases the SOP size up to 10% compared to the BDD-based method. Since the results for the SAT-based method are obtained by using several different options, Table 1 shows, in the column "#Cubes", the number of benchmarks for which the smallest SOP, in terms of number of cubes, was generated with each combination of options. We can notice that, for 36 benchmarks we get the smallest SOP when the canonical option is activated, thus for 68% of the benchmarks the canonical SOPs are smaller than the non-canonical ones.

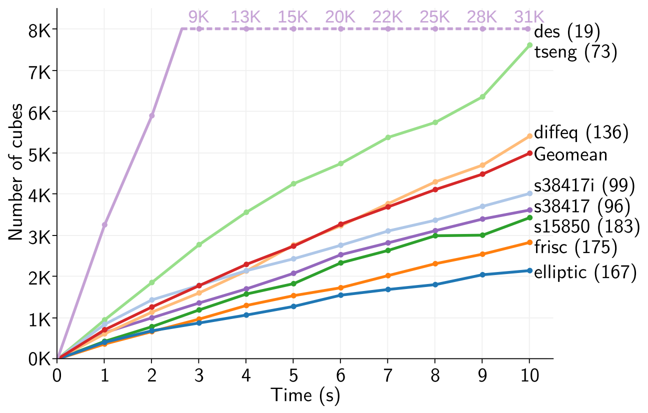

Figure 3: **Number of generated cubes for a partial SOP when the time limit is set between 1 and 10 seconds. The number of generated cubes depends on the size of the support set of the function, which is given in brackets. Only for the benchmark des we managed to generate complete SOPs for 3 to 25 outputs, in the given limits. For the other benchmarks, the generated cubes belong to one output.**

In terms of scalability, as Table 2 shows, the technique for isomorphic circuits presented in Section 3.4 allows computing an SOP only for 39.7% of the combinational outputs, each of them representing one isomorphic class, while for the other outputs we duplicate the generated SOP of the class representative. This reduces the runtime of our algorithm, and for benchmarks rich in isomorphic outputs, the proposed method is significantly faster than the BDD-based one. For example, the maximum speedup is achieved for the s35932 benchmark from the ISCAS'89 set, for which we generate SOPs only for 10 out of 2048 combinational outputs and thus, on average, the SAT-based method requires 0.06 seconds, while the BDD-based method finished in 1.83 seconds. However, on average, our SAT-based method is 8x slower than the BDD-based method for the public benchmarks. We have observed that the functions for expanding minterms to cubes are the bottleneck. For example, for the LT-DEC benchmarks, on average, 84% of the runtime is spent on this operation, while 8% is spent on minterm generation, 2% on removing redundant cubes, and 6% on other operations, such as dividing the outputs into classes, generating CNF, initializing SAT solver instances, etc.

Additionally, we conducted several experiments on industrial benchmarks, which could not be quoted in the paper. Our conclusion is that the SAT-based method is often as fast the BDD-based one and is definitely more scalable, that is, it can finish on some test-cases where the BDD-based method fails. We believe that the increased scalability is largely due to the fact that most of the industrial test cases have hundreds of inputs and outputs, which makes constructing global BDDs in the same manager problematic for all outputs at once. Our SAT-based method does not suffer from this limitation, because it computes the SOPs one by one, for each output separately. It can be argued that the BDD-based computation can also be performed on a per-output basis. However, in this case, the BDD manager will inevitably find different variable orders for different outputs, which will substantially reduce the quality of factoring. This

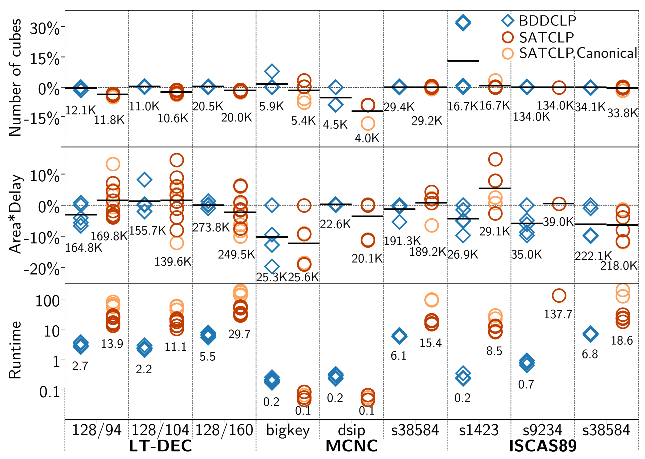

Figure 4: Variability of the results of the BDD-based and SAT-based method for the three largest circuits of each set. The results for the number of cubes and the area-delay product are normalized to one of the seeds of the BDD-based method. The horizontal lines show the averages for each method. Below the symbols are the actual minimal results.

is because factoring benefits from computing BDD-based SOP using the same variable order, which facilitates creating similar combinations of literals in different cubes, which in turn helps improve the quality of shared divisor extraction and factoring.

Finally, Figure 3 shows the number of generated cubes when we generate non-canonical SOPs and the time limit is set to t seconds, where t is an integer value such that $1 \leq t \leq 10$. For functions with larger support set, we usually generate less cubes as more time is required for cube expansion. Since we generate cubes progressively, unlike the BDD-based method, we can build partial SOPs even for large circuits, and these can be used for incremental applications. For this experiment, we are still generating both the on-set and the off-set SOP at the same time. However, for incremental application, we can generate just one of them, which would increase the number of generated cubes for a given time limit.

4.3 Case-Study: SAT-based SOPs for Generation of Multilevel Implementation

As explained in Section 4.2, we generate several SOPs with each method. Consequently, as Figure 4 shows, the different SOPs from each method, which have different size, result in multi-level networks with different area and delay. Figure 5 shows that our algorithm obtains Pareto optimal solutions compared to the BDD-based method for most benchmarks, if for each method we isolate the best circuit structures in terms of area-delay product. Table 1, with the column "A·D", shows the number of benchmarks with the smallest area-delay product generated with each combination of options. For 17 benchmarks we get the best results when the canonical option is activated. Thus, for 38% of the benchmarks generating canonical SOPs improves the results. Similarly, using the initial order of the primary inputs results in the best area-delay product for 91% of the circuits, and for 62% of them it is best to keep the order unreversed.

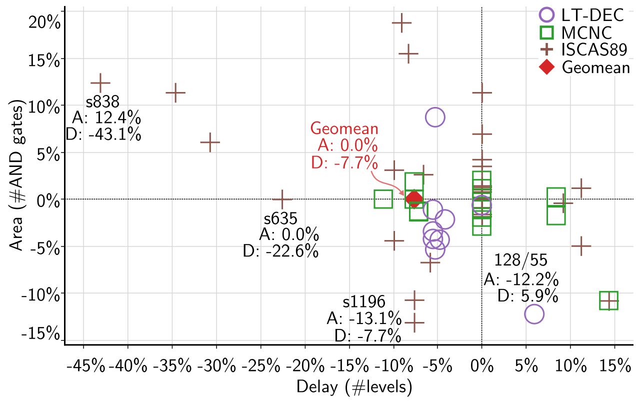

Figure 5: The best results for each benchmark after a multi-level description is built from SOPs generated by our SAT-based algorithm, compared to using a BDD-based SOPs. For most benchmarks, we obtain Pareto optimal solutions.

5. RELATED WORK

After the Quine-McCluskey algorithm [13], many algorithms and heuristics for SOP generation and minimization have been developed. Prior research largely divides into two main classes; BDD-based and ESPRESSO style algorithms.

BDD-based techniques, such as that of Minato-Morreale [15] and SCHERZO [5], first build a BDD or a ZDD to generate an SOP for a given Boolean function, and then apply a heuristic approach to minimize the BDD/ZDD size, which leads to a smaller SOP. If building a BDD is feasible, then an SOP, even a suboptimal one, can be generated. However, for some logic circuits with hundreds of inputs, building BDDs is sometimes not feasible or requires very long runtimes. On the other hand, given a time limit, our method returns a partial SOP when a complete SOP is not possible to obtain. This helps predicting if an SOP can be generated in a short runtime or a long runtime is required. Moreover, it allows to estimate the SOP size and if it is suitable or not for subsequent phases.

A second group of researchers had been inspired by the logic minimizer ESPRESSO [3] to develop fast and efficient logic minimization tools, such as ESPRESSO-MV [22]. Although ESPRESSO style techniques avoid the memory explosion problem of BDDs, they still incur impractical runtimes for very large Boolean functions. A recent work by Sapra et al. [23] proposed using a SAT solver to implement some of ESPRESSO's operators as a way to speed it up. However, an ESPRESSO based approach requires as input a computed SOP, on which its runtime and end results significantly rely. Contrary, our SAT-based algorithm generates and minimizes cubes one by one, and it returns irredundant SOP in relatively short runtime.

6. CONCLUSION

In this paper, we present a novel algorithm for progressive generation of irredundant canonical SOPs using heuristics based solely on SAT solving. Besides generating SOPs, the canonicity and the progressive generation make our heuristics desirable in many other areas where minterms or cubes are required, and for which the existing methods are either unscalable or impractical to use.

Regarding the quality of results, we show that for computing a complete SOP, on average, the SAT-based computation is as good as the BDD-based method. Moreover, the multi-level circuit structures derived using the SOPs generated by our approach are most often better or Pareto optimal.

Regarding the runtime, the proposed method is somewhat slower than the BDD-based method for most of the public benchmarks, but it is faster for circuits that are rich in isomorphic outputs. Other industrial benchmarks, which we tried but could not use in the paper, show that our method is both faster and more scalable, and therefore a good candidate for global circuit restructuring at least in that particular industrial setting.

The proposed method can also benefit from the ongoing improvement of modern SAT solvers. For example, recently we explored a new push/pop interface for assumptions used in the incremental SAT solving, which led to additional runtime improvements. As we show, for some circuits the results can improve by changing the variable order in which the cubes are expanded, but a careful study of this problem is required to improve further the quality of results.

In addition to runtime improvements, future work will focus on developing a dedicated SAT-based multi-output SOP computation, which computes cubes that are shared between several outputs. A recent publication [10] indicates that a significant improvement in quality (more than 10%) can be achieved by computing and factoring multi-output SOPs. We are not aware of a practical method for BDD-based multi-output SOP computation, so it is likely that SAT will be the only way to work with multiple outputs. Other directions of future work will include exploring the benefits of the progressive generation of canonical minterms and cubes in different areas. One such area is multi-level logic synthesis where incremental SAT-based decomposition methods can be developed based on partial SOPs computed for the output functions.

7. REFERENCES

[1] The EPFL Combinational Benchmark Suite, "Multi-output PLA benchmarks". `http://lsi.epfl.ch/benchmarks`.

[2] Berkeley Logic Synthesis and Verification Group, Berkeley, Calif. *ABC: A System for Sequential Synthesis and Verification.* `http://www.eecs.berkeley.edu/~alanmi/abc/`.

[3] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic, Boston, Mass., 1984.

[4] K. Chang, V. Bertacco, I. L. Markov, and A. Mishchenko. Logic synthesis and circuit customization using extensive external don't-cares. *ACM Trans. on Design Automation of Electronic Systems*, 15(3):26:1–24, May 2010.

[5] O. Coudert. Two-level logic minimization: An overview. *Integration, the VLSI journal*, 17(2):97–140, Oct. 1994.

[6] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, volume 2919, pages 502–18. Springer, May 2003.

[7] A. Ghosh, S. Devadas, and A. R. Newton. Test generation and verification for highly sequential circuits. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 10(5):652–67, May 1991.

[8] J.-H. R. Jiang, C.-C. Lee, A. Mishchenko, and C.-Y. R. Huang. To SAT or not to SAT: Scalable exploration of functional dependency. *IEEE Trans. on Computers*, C-59(4):457–67, Apr. 2010.

[9] D. E. Knuth. *Fascicle 6: Satisfiability*, volume 19 of *The Art of Computer Programming.* Addison-Wesley, Reading, Mass., Dec. 2015.

[10] V. N. Kravets. Application of a key-value paradigm to logic factoring. *Proceedings of the IEEE*, 103(11):2076–92, Nov. 2015.

[11] R.-R. Lee, J.-H. R. Jiang, and W.-L. Hung. Bi-decomposing large Boolean functions via interpolation and satisfiability solving. In *Proceedings of the 45th Design Automation Conference*, pages 636–41, Anaheim, Calif., June 2008.

[12] H.-P. Lin, J.-H. R. Jiang, and R.-R. Lee. To SAT or not to SAT: Ashenhurst decomposition in a large scale. In *Proceedings of the International Conference on Computer Aided Design*, pages 32–37, San Jose, Calif., Nov. 2008.

[13] E. J. McCluskey. Minimization of Boolean functions. *Bell System Tech. Journal*, 35(6):1417–44, Nov. 1956.

[14] K. L. McMillan. Interpolation and SAT-based model checking. In *Proceedings of the International Conference on Computer Aided Verification*, volume 2725, pages 1–13. Springer, July 2003.

[15] S. Minato. Fast generation of irredundant sum-of-products forms from binary decision diagrams. In *Proceedings of Synthesis And Simulation Meeting and International Interchange*, pages 64–73, Kobe, Japan, Apr. 1992.

[16] A. Mishchenko, R. Brayton, S. Jang, and V. N. Kravets. Delay optimization using SOP balancing. In *Proceedings of the International Conference on Computer Aided Design*, pages 375–82, San Jose, Calif., Nov. 2011.

[17] A. Mishchenko and R. K. Brayton. SAT-based complete don't-care computation for network optimization. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 412–17, Munich, Mar. 2005.

[18] A. Morgado and J. P. M. Silva. Good learning and implicit model enumeration. In *Proceedings of the 17th IEEE International Conference on Tools with Artificial Intelligence*, pages 131–36, Hong Kong, Nov. 2005.

[19] A. Nadel. Generating diverse solutions in SAT. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 287–301, Ann Arbor, Mich., June 2011.

[20] A. Petkovska, A. Mishchenko, M. Soeken, G. De Micheli, R. Brayton, and P. Ienne. Fast generation of lexicographic satisfiable assignments: Enabling canonicity in SAT-based applications. In *Proceedings of the 25th International Workshop on Logic and Synthesis*, Austin, Tex., June 2016.

[21] J. Rajski and J. Vasudevamurthy. The testability-preserving concurrent decomposition and factorization Boolean expressions. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 11(6):778–93, June 1992.

[22] R. L. Rudell and A. L. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 6(5):727–50, Sept. 1987.

[23] S. Sapra, M. Theobald, and E. M. Clarke. SAT-based algorithms for logic minimization. In *Proceedings of the 21st IEEE International Conference on Computer Design*, page 510, San Jose, Calif., Oct. 2003.

[24] G. S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, Symbolic Computation, pages 466–83. Springer, Berlin, 1983.

[25] A. K. Verma, P. Brisk, and P. Ienne. Iterative Layering: Optimizing arithmetic circuits by structuring the information flow. In *Proceedings of the International Conference on Computer Aided Design*, pages 797–804, San Jose, Calif., Nov. 2009.

[26] J. Yuan, A. Aziz, C. Pixley, and K. Albin. Simplifying Boolean constraint solving for random simulation-vector generation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 23(3):412–20, Mar. 2004.