Application of Logic Synthesis to Accelerate SAT-solving

Jan Kimr, Petr Fišer
Department of Digital Design, Faculty of Information Technology
Czech Technical University in Prague
Technická 9, Prague, Czech Republic
kimrjan@fit.cvut.cz, fiserp@fit.cvut.cz

***Abstract*—This paper presents a thorough analysis of the possibility of using standard logic synthesis and optimization to speed up solving satisfiability problem (SAT) instances coming from both the standard SAT benchmarks and different practical EDA applications. In principle, a SAT instance can be understood as a logic function described in a product-of-sums (POS) form. As such, it can be preprocessed by logic optimization prior to SAT solving. Logic optimization should reduce the instance (POS) size, making it "smaller". However, this does not imply being simpler for SAT solving. The second aspect is that logic optimization (resynthesis) could "break" some structures in the original SAT instance that made the instance difficult to solve. We try to investigate these two aspects and evaluate the efficiency of employing logic synthesis and optimization in SAT solving. We show that logic synthesis can both positively and negatively influence the time of SAT solving, mostly depending on the instance type. Finally, we will present some recommendations and a simple adaptive SAT-solving strategy. To our knowledge, this is the first study exploring the possibilities of using logic synthesis in SAT solving in such a complex way.**

***Index Terms*—logic synthesis, SAT, ATPG**

I. Introduction

With the beginning of this century, very fast and efficient Boolean Satisfiability Problem (SAT) solvers have been developed [1], [2], and this opened up a way of efficiently exploiting them in many different application areas. One of the probably most striking examples of this fact are Electronic Design Automation (EDA) tools, where SAT or its derivatives like pseudo-Boolean optimization (PBO) [3] or Satisfiability Modulo Theory (SMT) [4] are widely used. Actually, SAT has been used in model checking even before the advent of efficient SAT-solvers, and now it is used in modern model-checking engines [5]–[7]. The same holds for the Automated Test Patterns Generation process (ATPG) [8]–[14]. Recently, SAT, PBO, and SMT became a part of the state-of-the-art logic synthesis processes [15]–[24], aspiring to be part of industrial EDA tools [25].

Even though present SAT-solvers [1], [26]–[28] are meant to be efficient enough, their execution still takes a long time for some problem instances. Especially in the case of optimum circuits generation [19]–[22], large and difficult-to-solve instances occur. Next, in SAT-based ATPGs, difficult instances occur because of undetectable or hard-to-detect faults. Thus, one may ask whether the SAT-solvers task can be simplified by some means in order to improve the run time.

Since these SAT instances come from EDA, they are mostly constructed from logic circuits, i.e., netlists of gates. In particular, these problems involve determining the satisfiability of SAT instances obtained from artificially generated hardware (*conceptual hardware, miter*), and thus the problem solved is called a *circuit SAT*. In principle, the SAT instance is obtained by concatenating characteristic functions of all gates in the netlist, i.e., the final CNF is obtained by applying the Tseitin transformation [29]. For details, see, e.g., [8], [30]. Therefore, a straightforward way of simplification of the SAT instances can be offered: using logic optimization.

The most commonly used SAT instance representation is the Conjunctive Normal Form (CNF), i.e., the CNF-SAT problem is solved. This CNF can be understood as a logic function described in a product-of-sums (POS) form, which can be easily minimized by logic optimization tools [31], hoping that the resulting SAT instance would be "easier" for the SAT solver. Moreover, in some cases, the optimization can be performed even before generating the CNF from the miter. The ATPG process from [8] is such a case.

Several studies dealing with the application of logic synthesis in the SAT-solving process have been published [30], [32]. However, no definite conclusion can be derived from them since the sets of tested instances were rather limited.

In this paper, we present three case studies where logic optimization could be leveraged to possibly speed up the SAT-solving process. From the obtained results, we attempt to make a recommendation for using (or not using) logic optimization prior to the SAT-solving process. Both the logic optimization and SAT-solving time must be taken into account. Thus, our goal is to find a trade-off minimizing the overall run time.

Since the paper is mostly based on experimental work, its structure follows this idea. Thus, after a brief review of the related work in Section II, we immediately skip to the experimental part in Section III and in Section IV, we propose an adaptive mechanism for time-efficient SAT-solving. From the obtained results, we attempt to derive some general recommendations for processing SAT instances in Section V. Section VI concludes the paper.

Summarized, the main contributions of this paper are:

- Application of different logic optimization processes to SAT instances is thoroughly studied, for four state-of-the-art SAT solvers and three SAT instances sources.

- Detailed evaluation of the effects of logic optimization is made, and general recommendations for speeding up SAT-solving are proposed.
- An adaptive method to optimize the overall SAT-solving time is proposed.

II. Related Work and Motivation

Many simple CNF simplification techniques were already proposed years ago [33]. These techniques apply local transformations to the very SAT instances, trying to reduce the number of variables and clauses. However, later it was found that reducing the CNF instance size needs not reduce the SAT-solving time – just the contrary could happen and, conversely, increasing the SAT instance size by introducing more information helps significantly [8], [9], [34]. Actually, some of these SAT enhancements are contradictory to the CNF simplification techniques – some newly introduced variables and clauses could be eliminated by resolution, however, with a negative impact on the SAT solving time.

In contrast to these techniques, we aim at leveraging logic synthesis to *restructure* and possibly *minimize* the SAT instances and thereby help the SAT-solver in this way. One of the motivations was the fact that SAT instances obtained from circuits containing many XOR gates are hard for standard SAT-solvers [35], [20]. We hope that logic synthesis may "dissolve" these XORs and make the instances simpler for the SAT-solver.

There are some studies very similar to our work. In [30], the influence of running logic synthesis and optimization on the overall SAT-solving time was examined. Several SIS [36] scripts were applied to SAT instances coming from solving the integer factorization problem. These instances were constructed as conceptual hardware, i.e., logic circuits solving the problem were constructed and then transformed to SAT instances by the Tseitin transformation [29]. These were then solved by zChaff [2]. The observations published in that paper were actually the main motivation for our work. This is, (1) logic synthesis is orthogonal to SAT solving, (2) logic synthesis does help, (3) just simple and fast synthesis scripts suffice to achieve a great SAT-solving speed-up.

In [32], the authors study the influence of using simple ABC [31] commands to preprocess SAT instances obtained from bounded model checking (BMC) and problems coming from the *SAT-Race 2006* competition. The conclusions drawn were basically the same: for large circuit-SAT instances, preprocessing by logic optimization helps. However, for some SAT-Race competition problems, they have observed a slowdown. The SAT solver used was MiniSat [1].

Both above-mentioned papers document the same fact: circuit-SAT instances, i.e., SAT instances constructed from conceptual hardware by the Tseitin transformation [29], can be easily simplified by logic optimization, and as a result, the overall solving time can be reduced.

In this paper, we go into more depth. We explore the applicability of different logic optimization processes to different kinds of SAT instances and different state-of-the-art SAT solvers. Naturally, for small and easy-to-solve SAT instances, applying logic optimization makes no sense, and it would definitely even increase the overall run-time. Applying "powerful" optimization has been shown to be inefficient as well, as the time spent for optimization is not compensated by the SAT-solving time reduction [30].

Therefore, we ask *what kind of optimization* pays off and *what are the SAT instances* where the optimization pays off, for the "classical" MiniSat SAT solver [1] and the most modern SAT solvers Glucose [26], Kissat [27], and CaDiCaL [28]. This will document if things have changed since the previous analyses [30], [32] were published. From the obtained results, we attempt to propose general recommendations, ending up with an adaptive method taking advantage of the logic optimization most efficiently.

III. Experiments

In this section, we thoroughly evaluate three case studies: applying logic optimization to SAT instances obtained from (1) standard satisfiability benchmarks (SATLIB) [37], (2) instances obtained from SAT-based optimum circuits generator [20], and (3) SAT-ATPG instances generated by our in-house ATPG based on [8]. The influence of different logic optimization processes on the overall run time is studied.

A. Experimental Setup

The experiments were run on an Intel Xeon Gold 5218 (2.30 GHz) processor with 4 GB of memory running Debian 11 or Debian 12. All solvers were run using one CPU core. For practical reasons, the solvers had a time limit of 2000 seconds (real-time). All time values are CPU time unless specified otherwise.

We have exercised four SAT solvers: MiniSat 2.2 [1], Glucose-syrup 4.2.1 [26], Kissat 4.0.0 [27], and CaDiCaL 2.0.0 [28]. Glucose was run on four CPU cores as well, as this solver benefits mostly from its parallelism. However, since the results in the CPU time were almost the same as on one core, these results are not present in the tables.

Logic synthesis was performed by ABC [31]. The following ABC scripts were used, and they will be referred to in the subsequent text as follows:

- Just a simple conversion to an AND-inverter-graph (AIG) [31], without any optimization: *st*:
  `strash`
- A simple technology-independent optimization, with different efforts: *resyn2-[1, 2, 3]x*; the script repeated *x* times:
  `strash; resyn2`
- A powerful optimization and mapping into 2-input gates, with different efforts: *2-gate-[1, 5, 10, 15]x*; the script repeated *x* times:
  `&get -n; &st; &synch2; &if -m -a -K 2; &mfs -W 10; &st; &dch; &if -m -a -K 2; &mfs -W 10; &put`
- A powerful optimization and mapping into 6-input LUTs, with different efforts: *6-LUT-[1, 5, 10, 15]x*; the script repeated *x* times:

```
&get -n; &st; &synch2; &if -m -a -K 6;
&mfs -W 10; &st; &dch; &if -m -a -K 6;
&mfs -W 10; &put
```

In experiments where a SAT instance in a CNF was the source (i.e., Sections III-B and III-C), the CNF had to be transformed to BLIF [38] and then, after the optimization, converted back to a CNF. An in-house conversion tool was used for this purpose. The first conversion involved just a simple rewriting of the CNF to a two-level network of AND and OR gates. The latter conversion (i.e., where an optimized multi-level network is the input) was done by the Tseitin transformation [29]. As a result of this conversion, the number of variables significantly increases with respect to the original CNF since each signal in the optimized network is assigned a variable in the resulting CNF. However, as we will show here, the number of variables is not a crucial issue for SAT solvers.

In the case of SAT solving with synthesis, only the *synthesis time and subsequent SAT solving time are considered* and compared to solving without synthesis. The time of file conversion from CNF to BLIF and back is ignored, as the tool used for this conversion is not optimized for speed. However, both conversions can be done in a time linear with the instance size (the number of literals).

B. SATLIB benchmark

The SATLIB benchmark [37] offers a number of randomly generated hard 3-SAT instances, i.e., instances with the ratio of clauses to variables near 4.3 [39] and many other instances from different problem domains. For the experiments, uniform random 3-SAT instances with the numbers of variables and clauses equal to 200/860, 225/960, and 250/1065, as well as the pigeon-hole problem and Quasigroup (Latin square) instances were selected. All pigeon-hole problem instances are unsatisfiable; the uniform random 3-SAT sets and the Quasigroup set contain both satisfiable and unsatisfiable instances.

1) SAT-solvers comparison: In order to make a preliminary comparison of the SAT-solvers used, we have run them on the SATLIB benchmark and measured the average run times of solving the individual sets of instances. As the sets contain instances of approximately the same size, presenting average computation times is relevant enough. The only exception was the `hole10` instance of the pigeon-hole problem, which is much larger than the others and even wasn't solved by GLUCOSE within the time limit. Thus, it has been removed from this comparison. All the other instances were solved within the time limit by all SAT solvers.

The results are shown Table I. Surprisingly, the winner is the "old" MINISAT, followed by the newest solvers KISSAT and CADICAL, see the "Geomean" row indicating the geometric mean of the run times. The "count" column indicates the numbers of instances from the respective categories.

2) Identifying unsatisfiable instances: An unsatisfiable SAT instance means that it always evaluates to zero. Thus, when submitted to logic synthesis, a zero constant should be produced as a result. Consequently, a SAT-solver is no longer needed; the instance is "solved" by a mere logic synthesis.

TABLE I
COMPARISON OF SAT-SOLVERS (SATLIB)

		Run time [s]			
Benchmark / Solver	count	MINISAT	GLUCOSE	KISSAT	CADICAL
3-SAT 200/860 - SAT	100	0.03	1.53	0.14	0.16
3-SAT 200/860 - UNSAT	100	0.21	6.83	0.49	0.47
3-SAT 225-960 - SAT	100	0.24	6.16	0.28	0.34
3-SAT 225-960 - UNSAT	100	1.41	36.71	1.74	1.70
3-SAT 250-1065 - SAT	100	0.53	26.83	0.47	0.54
3-SAT 250-1065 - UNSAT	100	4.25	88.93	4.49	4.57
Quasigroup - SAT	10	0.16	1.38	0.29	0.09
Quasigroup - UNSAT	12	0.43	5.19	0.73	0.41
Pigeon-hole - UNSAT	4	0.50	44.64	0.13	1.06
Geomean		0.38	10.68	0.50	0.55

However, the synthesis must be "strong enough" to succeed. For this purpose, we have exercised all the tested synthesis processes (see Subsection III-A) and *unsatisfiable* instances from the above-mentioned benchmark sets. The results are shown in Table II.

TABLE II
PERCENTAGE OF UNSATISFIABLE INSTANCES IDENTIFIED BY SYNTHESIS (SATLIB)

Benchmark	200/860	225/960	250/1065	Quasigroup	Pigeon-hole
# of instances	100	100	100	12	5
st	0.0%	0.0%	0.0%	0.0%	0.0%
resyn2-1x	0.0%	0.0%	0.0%	0.0%	0.0%
resyn2-2x	0.0%	0.0%	0.0%	0.0%	0.0%
resyn2-3x	0.0%	0.0%	0.0%	0.0%	0.0%
2-gate-1x	0.0%	0.0%	0.0%	16.7%	80.0%
2-gate-5x	0.0%	0.0%	0.0%	16.7%	100.0%
2-gate-10x	0.0%	0.0%	0.0%	16.7%	100.0%
2-gate-15x	0.0%	0.0%	0.0%	33.3%	100.0%
6-LUT-1x	98.0%	37.0%	2.0%	16.7%	100.0%
6-LUT-5x	100.0%	50.0%	8.0%	33.3%	100.0%
6-LUT-10x	100.0%	64.0%	13.0%	33.3%	100.0%
6-LUT-15x	100.0%	67.0%	18.0%	41.7%	100.0%

We can see that only the most powerful optimization scripts were able to minimize the functions down to a constant, and this holds for simpler instances only. Thus, attempting to solve SAT by logic synthesis is probably not a good choice for these benchmarks.

3) How much synthesis helps: Having hints from previous studies showing a potential speedup when synthesis is used for SAT instances preprocessing [30], [32], we will investigate this in detail. Particularly, we have solved the SATLIB instances by the individual SAT-solvers, once alone and then preprocessed by synthesis. We have measured the total run times and calculated the improvements. The results are shown in Table III. The values indicate the average ratios of the run times without and with synthesis, i.e., the SAT-solving time of the original instance vs. the preprocessing time + the SAT-solving time of the optimized instance, for all instances in the given set. Thus, values above 1 indicate cases where using synthesis *for all the respective instances* pays off. These cells are highlighted in green. On the contrary, values approaching 0.00 indicate a large slowdown caused by synthesis.

From these preliminary results, we can see a sorry fact that *synthesis mostly does not help*. The exceptions are the

unsatisfiable pigeon-hole instances identified by synthesis (see Table II) and fast synthesis applied in combination with slower SAT-solvers – see the "Average time" row, which just copies the data from Table I, whereas the satisfiable and unsatisfiable instances are not distinguished.

4) How often synthesis helps: Based on the unlucky results from the previous experiment, we will investigate *how often* logic synthesis could help. Thus, we have solved all the instances with and without logic synthesis and calculated the percentage of instances that were solved faster when preprocessed by logic synthesis. The results are shown in Table IV.

We can see that a speedup was achieved mainly with the simplest (and thus fastest) synthesis scripts (*st* and *resyn2-[1, 2, 3]*) applied to the largest instances. One exception is the unsatisfiable Pigeon-hole instances, where the synthesis rendered them to constants, see Table II.

5) Potential time reduction: Finally, we will investigate how much the synthesis *could potentially help*. For this purpose, we solved each instance without and with synthesis and took the better time of these two. This value was then compared to the SAT solving without synthesis. The average results are shown in Table V. The "0%" entries indicate that using logic synthesis does not pay off for any of the instances from the respective benchmark set and SAT solver.

These values just show how large improvement *can be potentially achieved*, e.g., by running the two processes (synthesis + SAT and SAT only) in parallel. Moreover, this gives us a hint that if some *adaptive strategy* recognizing instances that should be processed by logic synthesis were devised, how large the achieved improvement could be.

We can observe the same phenomenon as in Table IV: only the simplest synthesis processes help, except for the unsatisfiable Pigeon-hole instances. However, the improvement is quite significant, especially for slower SAT-solvers – see the "Average time" row.

C. MinCirc instances

For this experiment, the tested SAT instances were produced by a SAT-based optimum circuits generator MinCirc [20]. These instances were generated when designing different optimal four- or five-variable functions implementations composed of 2-input AND and XOR gates, with a preference for XOR gates. There were 2,195 SAT instances generated when designing 220 four-variable functions. As for the five-variable functions, 3,076 SAT instances were generated, for 249 functions. In total, 5,271 SAT instances were obtained. The instances are split into three similarly sized groups based on the clauses count.

1) SAT-solvers comparison: The run times of the four SAT-solvers applied to the MinCirc instances will be measured here. The results are shown Table VI. Here it seems the newer SAT-solvers (KISSAT and CADICAL) are the fastest.

2) Identifying unsatisfiable instances: There are many unsatisfied instances in this set, particularly 3,292 out of 5,271 (62,5%). Similarly to Sec. III-B2, we will explore how well synthesis identifies them by rendering them to constant zeros. The results are shown in Table VII. We can see that the simplest processes (*st*, *resyn2*) could not optimize the instances down to zero; more powerful optimization was needed, and still it wasn't sufficient enough to cope with the largest instances.

3) How much synthesis helps: As in Subsection III-B3, we will evaluate the average speedup achieved when synthesis is *always* used, i.e., the preprocessing is made for all instances. The results are shown in Table VIII. We can see here that preprocessing all instances from a given class pays off only rarely, for the simplest optimization processes and the class of the largest instances. Thus, some mechanism selecting instances to be preprocessed is definitely needed here.

4) How often synthesis helps: Similarly to the experiment in Subsection III-B4, we have observed in how many cases the synthesis helped to decrease the solving time. The results can be found in Table IX. We can see that the results are much more positive than those in Table IV.

5) Potential time reduction: Finally, we have calculated the potential speedups by synthesis, as in Sec. III-B5. The results are shown in Table X. Again, more promising results can be seen here, compared to the SATLIB benchmarks.

We can see that the highest potential speedups were achieved by the simplest scripts (*st*, *resyn2-1x*) or by running the powerful scripts for one iteration only (*2-gate-1x*, or *6-LUT-1x*). Thus, the results suggest that incorporating even the simplest synthesis into the solving process could lead to a shorter solving time.

D. ATPG instances

For this set of experiments, we have used instances from a simple in-house SAT-based ATPG based on the very original idea of Larrabee [8]. In simple words, a conceptual hardware (*miter*) is generated by XORing the fault-free and faulty circuit. This miter is then converted to a CNF by the Tseitin transformation [29] and a test vector is generated as a satisfiability proof. It is possible to use synthesis to optimize the miter before doing the Tseitin transformation to make the CNF smaller. No CNF to BLIF conversion is needed in this case since the miter can be dumped to BLIF and optimized by logic synthesis (ABC). Thus, the synthesis can be directly incorporated into the ATPG process.

The ATPG process was run on circuits from a mixture of logic synthesis and testing benchmarks [40]. For each circuit, one SAT instance per tested fault was generated. These instances were mostly satisfiable; the unsatisfiable instances correspond to undetectable faults, which are relatively rare. These SAT instances were solved by the MINISAT solver only.

The circuits are divided into groups based on the solving time without synthesis in seconds. The results are reported for a circuit as a whole, i.e., solving SAT instances for all tested faults together.

1) Potential time reduction: Similarly to the previous subsections (III-B5, III-C5), we have measured the potential time reduction when synthesis is used. The results are shown in Table XI. Most of the ATPG instances were solved under

TABLE III
SATLIB BENCHMARKS: RUN TIME IMPROVEMENT BY SYNTHESIS

Benchmark	3-SAT 200/860				3-SAT 225/960				3-SAT 250/1065				Quasigroup				Pigeon-hole			
Solver	MiniSat	Glucose	Kissat	CaDiCaL	MiniSat	Glucose	Kissat	CaDiCaL	MiniSat	Glucose	Kissat	CaDiCaL	MiniSat	Glucose	Kissat	CaDiCaL	MiniSat	Glucose	Kissat	CaDiCaL
Average time [s]	0.1	4.4	0.3	0.3	0.6	19.2	0.8	0.9	2.7	61.5	2.5	2.9	0.3	3.5	0.5	0.3	25.4	44.6	0.6	12.3
st	0.22	0.94	0.89	0.75	0.49	1.71	1.28	0.95	0.50	2.58	1.34	1.06	0.07	0.23	0.27	0.07	31.22	1.25	1.25	1.69
resyn2-1x	0.18	0.71	0.68	0.59	0.39	1.88	0.99	0.82	0.59	2.09	0.98	1.00	0.02	0.15	0.06	0.02	14.76	2.07	0.20	1.34
resyn2-2x	0.17	0.63	0.57	0.54	0.35	2.52	0.84	0.74	0.46	1.48	0.96	0.99	0.01	0.11	0.04	0.02	15.40	1.55	0.20	1.14
resyn2-3x	0.15	0.65	0.52	0.49	0.34	1.32	0.71	0.71	0.44	2.58	0.89	0.90	0.01	0.10	0.03	0.01	15.26	1.81	0.23	1.21
2-gate-1x	0.03	0.40	0.09	0.09	0.11	0.85	0.18	0.19	0.22	1.19	0.30	0.34	∼0.00	0.02	∼0.00	∼0.00	0.58	42.35	0.19	1.11
2-gate-5x	0.01	0.23	0.03	0.03	0.04	0.57	0.06	0.07	0.11	0.78	0.13	0.15	∼0.00	0.01	∼0.00	∼0.00	0.54	41.93	0.18	1.07
2-gate-10x	0.01	0.16	0.02	0.02	0.02	0.39	0.04	0.04	0.07	0.62	0.08	0.09	∼0.00	0.01	∼0.00	∼0.00	0.52	40.85	0.17	1.03
2-gate-15x	∼0.00	0.12	0.01	0.01	0.02	0.33	0.02	0.03	0.05	0.56	0.06	0.07	∼0.00	∼0.00	∼0.00	∼0.00	0.53	40.78	0.17	1.04
6-LUT-1x	0.06	2.13	0.15	0.14	0.02	0.64	0.03	0.03	∼0.00	0.04	∼0.00	∼0.00	0.01	0.03	0.01	∼0.00	0.70	27.09	0.15	0.56
6-LUT-5x	0.06	2.11	0.15	0.14	0.02	0.63	0.03	0.03	∼0.00	0.01	∼0.00	∼0.00	∼0.00	0.01	∼0.00	∼0.00	0.69	26.53	0.14	0.55
6-LUT-10x	0.06	2.10	0.15	0.13	0.02	0.62	0.03	0.03	∼0.00	∼0.00	∼0.00	∼0.00	∼0.00	0.01	∼0.00	∼0.00	0.69	26.58	0.14	0.54
6-LUT-15x	0.06	2.09	0.15	0.13	0.02	0.62	0.03	0.03	∼0.00	∼0.00	∼0.00	∼0.00	∼0.00	0.01	∼0.00	∼0.00	0.68	25.72	0.12	0.52

TABLE IV
SATLIB BENCHMARKS: PERCENTAGE OF INSTANCES WHOSE SOLVING TIME DECREASED BY USING SYNTHESIS

Benchmark	3-SAT 200/860				3-SAT 225/960				3-SAT 250/1065				Quasigroup				Pigeon-hole			
Solver	MiniSat	Glucose	Kissat	CaDiCaL	MiniSat	Glucose	Kissat	CaDiCaL	MiniSat	Glucose	Kissat	CaDiCaL	MiniSat	Glucose	Kissat	CaDiCaL	MiniSat	Glucose	Kissat	CaDiCaL
st	0.5%	26.6%	14.1%	14.1%	6.0%	39.5%	33.0%	23.5%	7.5%	55.0%	43.0%	38.0%	0.0%	0.0%	4.5%	0.0%	80.0%	60.0%	60.0%	80.0%
resyn2-1x	0.0%	26.6%	9.5%	8.5%	3.0%	45.0%	25.5%	21.0%	10.5%	49.0%	33.5%	30.5%	0.0%	0.0%	0.0%	0.0%	60.0%	80.0%	0.0%	60.0%
resyn2-2x	0.0%	24.1%	5.0%	6.5%	2.0%	42.0%	23.5%	18.5%	6.5%	45.5%	35.0%	25.0%	0.0%	0.0%	0.0%	0.0%	60.0%	60.0%	0.0%	40.0%
resyn2-3x	0.0%	27.1%	3.0%	2.5%	2.5%	40.5%	17.5%	15.0%	4.0%	50.5%	30.5%	26.5%	0.0%	0.0%	0.0%	0.0%	60.0%	40.0%	0.0%	60.0%
2-gate-1x	0.0%	10.1%	0.0%	0.0%	0.0%	28.5%	0.0%	0.0%	0.0%	40.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	20.0%	80.0%	0.0%	20.0%
2-gate-5x	0.0%	0.0%	0.0%	0.0%	0.0%	13.0%	0.0%	0.0%	0.0%	20.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	20.0%	80.0%	0.0%	20.0%
2-gate-10x	0.0%	0.0%	0.0%	0.0%	0.0%	2.0%	0.0%	0.0%	0.0%	11.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	20.0%	80.0%	0.0%	20.0%
2-gate-15x	0.0%	0.0%	0.0%	0.0%	0.0%	1.5%	0.0%	0.0%	0.0%	7.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	20.0%	80.0%	0.0%	20.0%
6-LUT-1x	0.0%	36.7%	0.0%	0.0%	0.0%	9.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	20.0%	80.0%	0.0%	20.0%
6-LUT-5x	0.0%	36.7%	0.0%	0.0%	0.0%	9.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	20.0%	80.0%	0.0%	20.0%
6-LUT-10x	0.0%	36.7%	0.0%	0.0%	0.0%	9.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	20.0%	80.0%	0.0%	20.0%
6-LUT-15x	0.0%	36.7%	0.0%	0.0%	0.0%	9.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	20.0%	80.0%	0.0%	20.0%

TABLE V
SATLIB BENCHMARKS: POTENTIAL SOLVING TIME REDUCTION

Benchmark	3-SAT 200/860				3-SAT 225/960				3-SAT 250/1065				Quasigroup				Pigeon-hole			
Solver	MiniSat	Glucose	Kissat	CaDiCaL	MiniSat	Glucose	Kissat	CaDiCaL	MiniSat	Glucose	Kissat	CaDiCaL	MiniSat	Glucose	Kissat	CaDiCaL	MiniSat	Glucose	Kissat	CaDiCaL
Average time [s]	0.1	4.4	0.3	0.3	0.6	19.2	0.8	0.9	2.7	61.5	2.5	2.9	0.3	3.5	0.5	0.3	25.4	44.6	0.6	12.3
st	0.1%	18.4%	6.7%	6.9%	3.2%	20.9%	11.3%	7.9%	1.7%	20.3%	12.8%	9.3%	0.0%	0.0%	1.5%	0.0%	99.1%	6.6%	49.4%	4.5%
resyn2-1x	0.0%	18.8%	4.1%	3.7%	2.5%	21.9%	10.2%	7.3%	3.4%	19.3%	10.7%	9.0%	0.0%	0.0%	0.0%	0.0%	98.1%	32.9%	0.0%	39.5%
resyn2-2x	0.0%	15.5%	1.2%	3.2%	1.3%	23.9%	8.9%	4.6%	1.0%	15.7%	9.9%	9.2%	0.0%	0.0%	0.0%	0.0%	98.2%	39.3%	0.0%	41.9%
resyn2-3x	0.0%	16.1%	1.4%	2.3%	0.7%	19.1%	5.8%	5.7%	1.8%	17.6%	9.9%	7.1%	0.0%	0.0%	0.0%	0.0%	98.2%	12.7%	0.0%	43.7%
2-gate-1x	0.0%	5.0%	0.0%	0.0%	0.0%	13.8%	0.0%	0.0%	0.0%	14.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.7%	98.9%	0.0%	4.8%
2-gate-5x	0.0%	0.0%	0.0%	0.0%	0.0%	7.1%	0.0%	0.0%	0.0%	7.6%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.7%	98.9%	0.0%	4.8%
2-gate-10x	0.0%	0.0%	0.0%	0.0%	0.0%	1.0%	0.0%	0.0%	0.0%	3.9%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.7%	98.9%	0.0%	4.7%
2-gate-15x	0.0%	0.0%	0.0%	0.0%	0.0%	0.7%	0.0%	0.0%	0.0%	3.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.7%	98.9%	0.0%	4.7%
6-LUT-1x	0.0%	64.3%	0.0%	0.0%	0.0%	5.8%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	60.8%	88.0%	0.0%	15.7%
6-LUT-5x	0.0%	64.2%	0.0%	0.0%	0.0%	5.8%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	60.0%	88.0%	0.0%	14.1%
6-LUT-10x	0.0%	64.1%	0.0%	0.0%	0.0%	5.8%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	60.8%	88.0%	0.0%	15.9%
6-LUT-15x	0.0%	64.0%	0.0%	0.0%	0.0%	5.8%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	60.8%	88.0%	0.0%	15.9%

5 seconds in which case there were almost no potential improvements. With the increasing size of instances, the potential improvements increased. The highest being 32.0% for *st* synthesis. Furthermore, using *st* synthesis to preprocess all miters created during solving would lead to a 19.7% decrease in solving time.

IV. ADAPTIVE SAT-SOLVING PROCEDURE

Preprocessing *all* SAT instances using synthesis would be beneficial only for SAT-ATPG instances with *st* synthesis. However, even in this case, synthesis would decrease the solving time of only 10.5% of circuits while increasing that of others. Therefore, it is reasonable to assume that selecting instances to preprocess could lead to better results.

A. Restarting

Based on the idea that instances solved fast usually do not benefit from the synthesis, we tested an approach considering this. We can attempt to solve an instance without synthesis for some time t. If it is not solved within this time, the process is ended, and the instance is preprocessed and solved again. We

TABLE VI
COMPARISON OF SAT-SOLVERS (MINCIRC)

# of clauses / Solver	count	Run time [s] MINISAT	GLUCOSE	KISSAT	CADICAL
[428, 4608) - SAT	77	0,02	0,16	0,05	0,03
[428, 4608) - UNSAT	1511	0,01	0,2	0,03	0,02
[4608, 12688) - SAT	688	1,37	3,54	0,69	0,67
[4608, 12688) - UNSAT	1197	7,08	21,75	1,58	1,61
[12688, 62392) - SAT	1124	60,99	238,65	19,02	36,4
[12688, 62392) - UNSAT	349	339,49	695,86	42,88	56,22
Geomean		1,84	8,59	1,07	1,06

TABLE VII
PERCENTAGE OF UNSATISFIABLE INSTANCES IDENTIFIED BY SYNTHESIS (MINCIRC)

# of clauses	[428, 4608)	[4608, 12688)	[12688, 62392)
# of instances	1,511	1,197	584
st	0.0%	0.0%	0.0%
resyn2-1x	0.0%	0.0%	0.0%
resyn2-2x	0.0%	0.0%	0.0%
resyn2-3x	0.0%	0.0%	0.0%
2-gate-1x	82.8%	24.6%	0.0%
2-gate-5x	88.3%	33.2%	0.0%
2-gate-10x	89.0%	36.8%	0.0%
2-gate-15x	89.3%	38.3%	0.0%
6-LUT-1x	89.3%	23.2%	0.0%
6-LUT-5x	89.5%	29.5%	0.0%
6-LUT-10x	89.7%	32.4%	0.0%
6-LUT-15x	90.0%	35.2%	0.0%

call this model *restarting* and it can be used on its own or in combination with some machine-learning (ML) model.

A downside of this approach is that it increases the time of solving the instances that were restarted and solved with synthesis by the time t.

B. Selecting instances to run with synthesis

A different approach is to try to predict whether to use synthesis based on some properties of an instance. There are two possibilities that we can predict: (1) we can predict one of two classes (binary classification), whether it is beneficial to use synthesis or not, or (2) real value (regression) corresponding to how sure it is that synthesis will improve the solving time. In the first case, synthesis is beneficial if $syn < nosyn$. In the second case, we take $log(\frac{nosyn}{syn})$ (both the divisor and whole fraction needs to be increased by a small constant, e.g., 10^{-9}, in order to avoid undefined operations). This value is zero if $syn = nosyn$, less than zero if $syn > nosyn$, and greater than zero $syn < nosyn$. The larger the absolute value of the logarithm is, the more sure the use or not use of synthesis is. It can also be mapped to $[0, 1]$ interval using sigmoid function ($\sigma(x) = \frac{1}{1+e^{-x}}$). In which case, the value can be interpreted as "*how high is the probability of synthesis speeding up the SAT-solving*".

In either case, the main problem is the imbalance in data – at best only around 10.5% of instances were sped up after using logic synthesis. This can be problematic as models can end up predicting the more frequent class in the case of classification or only values less than zero in the case of regression.

TABLE VIII
MINCIRC: RUN TIME IMPROVEMENT BY SYNTHESIS

# of clauses	[428, 4608)				[4608, 12688)				[12688, 62392)			
Solver	MiniSat	Glucose	Kissat	CaDiCaL	MiniSat	Glucose	Kissat	CaDiCaL	MiniSat	Glucose	Kissat	CaDiCaL
st	0.06	0.20	0.16	0.05	0.64	0.82	0.71	0.40	1.23	1.32	1.67	1.29
resyn2-1x	0.02	0.15	0.06	0.02	0.26	0.61	0.34	0.23	0.33	1.16	0.85	0.78
resyn2-2x	0.01	0.12	0.05	0.02	0.19	0.50	0.26	0.19	0.31	0.96	0.68	0.64
resyn2-3x	0.01	0.11	0.04	0.02	0.14	0.50	0.22	0.16	0.27	0.89	0.55	0.55
2-gate-1x	∼0.00	0.04	0.01	∼0.00	0.08	0.19	0.04	0.03	0.19	0.51	0.15	0.18
2-gate-5x	∼0.00	0.02	∼0.00	∼0.00	0.03	0.09	0.01	0.01	0.11	0.27	0.06	0.07
2-gate-10x	∼0.00	0.02	∼0.00	∼0.00	0.02	0.06	0.01	0.01	0.08	0.19	0.04	0.05
2-gate-15x	∼0.00	0.02	∼0.00	∼0.00	0.01	0.05	0.01	∼0.00	0.06	0.16	0.03	0.03
6-LUT-1x	0.01	0.05	0.01	0.01	0.14	0.25	0.07	0.06	0.32	0.54	0.18	0.21
6-LUT-5x	∼0.00	0.03	0.01	∼0.00	0.05	0.13	0.03	0.02	0.18	0.33	0.09	0.11
6-LUT-10x	∼0.00	0.03	0.01	∼0.00	0.03	0.09	0.02	0.01	0.13	0.25	0.06	0.07
6-LUT-15x	∼0.00	0.02	0.01	∼0.00	0.02	0.07	0.01	0.01	0.10	0.20	0.04	0.05

Multiple machine learning (ML) models were tried for selecting instances: linear regression, logistic regression, naive Bayes, k-nearest neighbors, random forest, and gradient boosted decision trees. If models have hyperparameters that can be tuned, multiple different values were tried.

C. Selecting instances for restarting

This approach combines both previously described ones. Firstly, an ML model is trained to predict whether to use synthesis for preprocessing. Then, the restarting model's t is found *only* on instances selected to be run with synthesis.

The final model predicts whether to restart (use synthesis) after t seconds of solving without synthesis. Instances that should not be restarted are solved without synthesis. Those that should be restarted are run without synthesis for time t and then restarted and run with synthesis, in case they have not already been solved.

The idea is that if the ML model falsely predicts to use the preprocessing for an instance that would be solved quickly without it, the restarting part fixes it in some cases. If the ML model always predicts the use of synthesis, this model transforms into the original restarting model; similarly, if the model's t is around 0, it transforms into the selecting model.

D. Model training and evaluation

Since the dataset is relatively small, instances sped up by synthesis are scarce, and improvements achieved using synthesis differ significantly; using train, dev, and test sets for evaluating models is not ideal as the results would strongly depend on how instances were divided. For this reason, a cross-validation (CV) will be used. This should help with this problem as the models are trained and evaluated multiple times on different parts of data and results are then averaged together.

In the case of the restarting model or if no hyperparameters of the model are tuned, a simple 5-fold CV is used. In other cases, nested CV is used. The outer CV is 5-fold and splits the test part of the data from the train and dev parts, and the inner CV is 4-fold and splits the train and dev parts. The inner CV selects the best parameters and the outer is used to measure the quality of the best parameters. The reported

TABLE IX

MINCIRC: PERCENTAGE OF INSTANCES WHOSE SOLVING TIME DECREASED BY USING SYNTHESIS. INSTANCES THAT WERE SOLVED NEITHER WITH NOR WITHOUT SYNTHESIS ARE LEFT OUT.

# clauses	[428, 4608)				[4608, 12688)				[12688, 62392)				all instances			
Solver	MiniSat	Glucose	Kissat	CaDiCaL	MiniSat	Glucose	Kissat	CaDiCaL	MiniSat	Glucose	Kissat	CaDiCaL	MiniSat	Glucose	Kissat	CaDiCaL
st	0.0%	2.4%	0.1%	0.0%	9.9%	21.3%	18.0%	5.4%	21.8%	35.6%	44.4%	31.6%	10.4%	19.7%	21.2%	11.9%
resyn2-1x	0.0%	0.7%	0.0%	0.0%	3.3%	15.4%	4.7%	2.0%	7.5%	30.3%	32.8%	25.8%	3.6%	15.3%	12.5%	8.9%
resyn2-2x	0.0%	0.3%	0.0%	0.0%	3.0%	12.2%	2.6%	1.3%	7.6%	31.7%	26.0%	24.6%	3.5%	14.5%	9.5%	8.3%
resyn2-3x	0.0%	0.1%	0.0%	0.0%	1.9%	12.2%	1.6%	1.2%	7.5%	29.0%	21.3%	20.8%	3.0%	13.6%	7.6%	7.1%
2-gate-1x	0.0%	0.0%	0.0%	0.0%	0.4%	0.9%	0.1%	0.3%	7.1%	14.9%–15.0%	7.0%–7.1%	7.7%	2.3%	5.0%–5.0%	2.3%–2.3%	2.5%
2-gate-5x	0.0%	0.0%	0.0%	0.0%	0.4%	0.4%	0.0%	0.1%	3.9%–4.0%	8.2%	2.7%–3.4%	3.3%–3.5%	1.4%–1.4%	2.7%	0.9%–1.1%	1.1%–1.1%
2-gate-10x	0.0%	0.0%	0.0%	0.0%	0.3%	0.2%	0.0%	0.0%	2.6%–2.7%	4.8%–5.1%	1.7%–2.8%	1.9%–2.1%	0.9%–0.9%	1.6%–1.6%	0.5%–0.9%	0.6%–0.7%
2-gate-15x	0.0%	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	1.9%–2.0%	3.7%–4.2%	1.3%–3.1%	1.2%–1.4%	0.6%–0.7%	1.1%–1.3%	0.4%–1.0%	0.4%–0.4%
6-LUT-1x	0.0%	0.0%	0.0%	0.0%	1.4%	2.5%	0.2%	0.5%	14.4%	17.9%	8.3%–8.5%	9.9%	4.9%	6.5%	2.8%–2.8%	3.3%
6-LUT-5x	0.0%	0.0%	0.0%	0.0%	0.6%	0.6%	0.0%	0.1%	10.2%	10.9%–11.0%	3.5%–3.7%	4.3%–4.4%	3.4%	3.6%–3.6%	1.1%–1.2%	1.4%–1.4%
6-LUT-10x	0.0%	0.0%	0.0%	0.0%	0.4%	0.4%	0.0%	0.1%	7.3%	8.4%–8.6%	3.3%–3.5%	3.1%–3.1%	2.4%	2.7%–2.8%	1.1%–1.2%	1.0%–1.0%
6-LUT-15x	0.0%	0.0%	0.0%	0.0%	0.3%	0.2%	0.0%	0.1%	5.4%–5.5%	6.0%–6.3%	2.5%–3.2%	2.1%	1.8%–1.8%	1.9%–2.0%	0.8%–1.0%	0.7%

TABLE X

MINCIRC: POTENTIAL SOLVING TIME REDUCTION. ONLY INSTANCES SOLVED BOTH WITH AND WITHOUT SYNTHESIS ARE CONSIDERED. THE AVERAGE TIME IS CALCULATED FROM INSTANCES SOLVED WITHOUT SYNTHESIS WITHIN THE TIME LIMIT.

# clauses	[428, 4608)				[4608, 12688)				[12688, 62392)				all instances			
Solver	MiniSat	Glucose	Kissat	CaDiCaL	MiniSat	Glucose	Kissat	CaDiCaL	MiniSat	Glucose	Kissat	CaDiCaL	MiniSat	Glucose	Kissat	CaDiCaL
Average time [s]	0.0	0.2	0.0	0.0	5.0	15.1	1.3	1.3	141.4	340.5	92.7	55.3	45.1	109.9	30.4	17.8
st	0.0%	3.2%	0.1%	0.0%	23.7%	7.6%	9.7%	9.2%	16.9%	19.7%	25.3%	37.3%	17.3%	19.0%	24.9%	36.5%
resyn2-1x	0.0%	1.2%	0.0%	0.0%	14.2%	7.3%	4.6%	12.5%	13.3%	17.2%	22.6%	36.2%	13.4%	16.6%	22.2%	35.5%
resyn2-2x	0.0%	0.3%	0.0%	0.0%	10.4%	5.8%	3.9%	8.8%	13.1%	17.6%	22.6%	37.8%	12.9%	16.9%	22.1%	37.0%
resyn2-3x	0.0%	0.1%	0.0%	0.0%	11.6%	7.0%	2.4%	12.1%	15.4%	17.3%	20.0%	32.9%	15.2%	16.7%	19.6%	32.3%
2-gate-1x	0.0%	0.0%	0.0%	0.0%	15.7%	1.9%	0.5%	4.0%	14.1%	11.6%	14.3%	22.5%	14.2%	11.0%	14.0%	21.9%
2-gate-5x	0.0%	0.0%	0.0%	0.0%	11.4%	2.0%	0.0%	0.6%	7.9%	7.1%	9.8%	12.5%	8.1%	6.8%	9.5%	12.1%
2-gate-10x	0.0%	0.0%	0.0%	0.0%	4.7%	0.9%	0.0%	0.0%	4.5%	4.3%	5.8%	9.0%	4.5%	4.1%	5.6%	8.7%
2-gate-15x	0.0%	0.0%	0.0%	0.0%	0.9%	0.0%	0.0%	0.0%	4.2%	2.7%	3.7%	4.9%	4.0%	2.5%	3.6%	4.7%
6-LUT-1x	0.0%	0.0%	0.0%	0.0%	24.0%	3.4%	1.1%	5.0%	20.7%	14.0%	17.6%	24.2%	20.8%	13.4%	17.1%	23.6%
6-LUT-5x	0.0%	0.0%	0.0%	0.0%	15.3%	1.6%	0.0%	3.0%	16.9%	9.7%	10.6%	18.3%	16.8%	9.2%	10.3%	17.9%
6-LUT-10x	0.0%	0.0%	0.0%	0.0%	9.8%	1.2%	0.0%	0.4%	13.5%	7.7%	9.2%	11.4%	13.4%	7.3%	9.0%	11.1%
6-LUT-15x	0.0%	0.0%	0.0%	0.0%	10.1%	0.5%	0.0%	0.4%	9.3%	5.5%	7.0%	9.2%	9.3%	5.3%	6.8%	8.9%

TABLE XI

ATPG CIRCUITS: POTENTIAL SOLVING TIME REDUCTION

Original time	[0, 5)	[5, 50)	[50, 100)	[100, 200)	[200, 500)	[500, 1000)	[1000, inf)	all circuits
# circuits	788	274	60	47	27	32	20	1248
st	0.5%	3.3%	7.5%	12.2%	9.3%	20.1%	51.2%	32.0%
resyn2-1x	0.2%	2.3%	3.5%	14.7%	8.7%	7.6%	50.0%	28.4%
resyn2-2x	0.2%	1.9%	3.1%	13.0%	7.0%	6.5%	48.8%	27.2%
resyn2-3x	0.2%	1.5%	2.4%	11.8%	5.9%	5.5%	48.6%	26.6%
2-gate-1x	0.0%	0.8%	1.5%	7.3%	1.2%	1.5%	41.8%	21.5%
2-gate-5x	0.0%	0.8%	1.2%	3.1%	5.7%	0.8%	40.7%	20.9%
2-gate-10x	0.0%	0.5%	0.9%	0.1%	0.0%	0.3%	39.6%	19.5%
2-gate-15x	0.0%	0.5%	0.6%	0.0%	0.0%	0.2%	38.4%	18.9%
6-LUT-1x	0.0%	1.0%	1.4%	5.4%	3.0%	1.0%	38.7%	19.9%
6-LUT-5x	0.0%	0.8%	1.3%	1.0%	2.8%	0.8%	39.0%	19.7%
6-LUT-10x	0.0%	0.6%	1.1%	0.0%	2.8%	0.6%	38.5%	19.3%
6-LUT-15x	0.0%	0.5%	0.9%	0.0%	2.8%	0.7%	38.1%	19.1%

results are averages from test parts, while the models with the best parameters are trained on the rest of the data.

Using simple metrics such as accuracy or F1-score to evaluate the correctness of predictions is not enough since the speedups or slowdowns of using synthesis differ significantly between instances – the goal is to correctly predict instances where using synthesis would lead to significant speedup or slowdown. One metric that takes this into consideration is the average change in solving time (Equation 1; the model is represented as a function predicting the solving time of each instance). Results below zero mean a speedup, while those above a slowdown. This metric is used for selecting the best model. Test parts used for model evaluation have different total times of solving instances without synthesis. Using a simple average to aggregate the results of this metric together would not take this into account, thus the weighted average and standard deviation are used with the total solving times without synthesis as the weights.

$$\frac{\sum_{(nosyn,syn,data)} model\,(nosyn, syn, data)}{\sum_{(nosyn,syn,data)} nosyn} - 1 \qquad (1)$$

Another presented metric is the ratio of instances whose solving time was the same or faster. This is complementary to the ratio of instances that were solved in a longer time due to the use of the model. In case solving of no instance was slowed down, and there is a speedup in solving time as well, using this model has no downside.

E. MinCirc instances

For selecting whether to use synthesis, properties of SAT instances used in SATZILLA solver [41] were used. These were extended with domain-specific properties as the MinCirc instances are generated sequentially: the number of SAT instance, whether the previous instance is satisfiable, and

TABLE XII
MINCIRC: BEST MODELS FOR EACH SOLVER AND SYNTHESIS WHICH ACHIEVED SPEEDUP ABOVE 5%

Solver	Synthesis	Name	Restarting after t	Total time change
CADICAL	*resyn2-2x*	Restarting	$127.9 \pm 4.4\%$	$-21.6 \pm 7.7\%$
CADICAL	*resyn2-1x*	Restarting	$128.5 \pm 3.0\%$	$-12.7 \pm 10.2\%$
CADICAL	*resyn2-3x*	Restarting	$205.8 \pm 191.3\%$	$-11.2 \pm 8.2\%$
CADICAL	*6-LUT-5x*	Restarting	$641.5 \pm 8.4\%$	$-7.2 \pm 8.5\%$
CADICAL	*2-gate-5x*	Restarting	$633.0 \pm 1.0\%$	$-6.2 \pm 5.9\%$
CADICAL	*6-LUT-1x*	Restarting	$633.0 \pm 1.0\%$	$-5.6 \pm 9.3\%$
MINISAT	*6-LUT-5x*	KNN (clasif.)[1]	-	$-5.6 \pm 10.0\%$
CADICAL	*2-gate-1x*	Restarting	$430.2 \pm 277.2\%$	$-5.3 \pm 16.3\%$
CADICAL	*st*	Gaussian NB	$212.9 \pm 235.1\%$	$-5.0 \pm 18.4\%$

TABLE XIII
ATPG: BEST MODEL FOR EACH SYNTHESIS

Synthesis	Name	Restarting after t	Total time change
st	Gaussian NB[1]	-	$-26.3 \pm 29.2\%$
resyn2-1x	KNN (clasif.)[2]	-	$-14.8 \pm 18.7\%$
resyn2-3x	KNN (clasif.)[2]	-	$-14.3 \pm 18.3\%$
resyn2-2x	KNN (clasif.)[2]	-	$-13.7 \pm 19.4\%$
6-LUT-10x	Logistic Regression	-	$-6.3 \pm 12.8\%$
2-gate-1x	KNN (clasif.)[2]	5.8 ± 7.9	$-6.2 \pm 9.8\%$
6-LUT-1x	KNN (clasif.)[2]	5.8 ± 7.9	$-6.2 \pm 9.8\%$
2-gate-10x	Logistic Regression	-	$-6.1 \pm 9.7\%$
6-LUT-15x	Logistic Regression[2]	-	$-6.1 \pm 12.5\%$
6-LUT-5x	KNN (clasif.)[2]	-	$-5.8 \pm 9.8\%$
2-gate-5x	KNN (clasif.)[2]	5.8 ± 7.9	$-5.8 \pm 9.8\%$
2-gate-15x	Logistic Regression[1]	-	$-5.5 \pm 10.7\%$

[1] Data were standardized
[2] Data were scaled to [0, 1] interval

whether it is from generating an optimal implementation of four or five variable function.

Results of the best models can be found in Table XII.

F. ATPG instances

In the case of ATPG instances, the decision to use or not use synthesis was made based on the properties of the circuit. These properties were: # of primary inputs, # of primary output, # of gates, # of edges, the total number of terms, the total number of literals, # of combinational levels, # of connected components, # of XOR gates ($xors$), # of gate equivalents, the maximum fan-in, the average fan-in, the maximum fan-out, the average fan-out. If the decision were to use synthesis preprocessing, each generated miter would be preprocessed before converting to CNF.

Results of the best models can be found in Table XIII.

V. DISCUSSION

Using preprocessing by logic synthesis was beneficial for some of the larger instances. For the small ones, their preprocessing was usually not beneficial. Synthesis could potentially significantly decrease the solving time of some instances, however, using it to preprocess all instances would in almost all cases increase the overall solving time.

An interesting result of using synthesis on smaller, *unsatisfiable* instances was that some syntheses transformed those instances into trivial ones (constant zero outputs). Nevertheless, this mostly did not lead to faster SAT solving.

When solving the SATLIB benchmark instances, the potential speedups were usually fairly small for the fastest solvers. The only exceptions were pigeon-hole instances where the speedups were much larger, mostly due to the discovery of unsatisfiable instances by logic synthesis.

Preprocessing MinCirc instances using logic synthesis was slightly more successful. From the group of the largest instances, more than one third of instances preprocessed with *st* script were solved faster by GLUCOSE, KISSAT, and CADICAL solvers. The individual instances that were sped up after preprocessing differed between the solvers – most being sped up when solved by just one solver – suggesting that there is no "type" of an instance that would benefit from synthesis preprocessing regardless of the solver.

When exploring the possibilities of incorporating synthesis into SAT-solving, using the *restarting* approach led to better results than selecting instances to preprocess using an ML model.

The *restarting* can also be used more generally than selecting as it requires data to train the ML model. So the key takeaway is that if an instance takes too long to solve, it might be beneficial to end the process and try to solve it preprocessed by logic synthesis. Devising an adaptive time threshold will be the topic of our future work.

VI. CONCLUSION

This paper presented an analysis of the possibility of using logic synthesis to preprocess SAT instances prior to their submission to a SAT-solver. Different ABC synthesis scripts were exercised, in combination with four state-of-the-art SAT solvers (MINISAT, GLUCOSE, KISSAT, CADICAL). We have observed that logic synthesis could help to reduce the solving time for some instances. Generally, we can conclude that the simplest (and thus fastest) synthesis processes, in combination with large instances and slow SAT-solvers, often help to speed up the overall solving time. On the other hand, powerful and time-consuming synthesis processes are often able to identify unsatisfiable SAT instances by rendering them to constant zero. However, some SAT solvers can often do the job faster.

Finally, adaptive mechanisms to decide whether to use preprocessing by logic synthesis were proposed. We have found that probably the best and simplest way to go is to run the SAT-solver on the original instance with a small time threshold, and when this threshold is exceeded, to preprocess the SAT instance by a simple optimization process and re-run the SAT-solving.

ACKNOWLEDGMENT

This work was supported by the Student Summer Research Program 2023 of FIT CTU in Prague and the "Advanced Chip Design Research Center (ACDRC), Embedded AI Processor for Automotive Applications", Act. No. 340/2015 Coll. project. Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

REFERENCES

[1] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, ser. Lecture Notes in Computer Science, E. Giunchiglia and A. Tacchella, Eds. Springer Berlin Heidelberg, 2004, vol. 2919, pp. 502–518.

[2] Y. S. Mahajan, Z. Fu, and S. Malik, "Zchaff2004: An efficient SAT solver," in *International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[3] N. Eén and N. Sörensson, "Translating pseudo-boolean constraints into SAT," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 1–26, 11 2006.

[4] C. Barrett and C. Tinelli, *Satisfiability Modulo Theories*. Cham: Springer International Publishing, 2018, pp. 305–343.

[5] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs," in *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, 1999, pp. 317–320.

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

[7] A. Biere and D. Kröning, *SAT-Based Model Checking*. Cham: Springer International Publishing, 2018, pp. 277–303.

[8] T. Larrabee, "Test pattern generation using Boolean satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 1, pp. 4–15, Jan. 1992.

[9] R. Drechsler, S. Eggersgl, G. Fey, and D. Tille, *Test Pattern Generation Using Boolean Proof Engines*, 1st ed. Springer Publishing Company, Incorporated, 2009.

[10] S. Eggersglüß, K. Schmitz, R. Krenz-Bååth, and R. Drechsler, "On optimization-based ATPG and its application for highly compacted test sets," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 2104–2117, 2016.

[11] H. Chen and J. Marques-Silva, "A two-variable model for SAT-based ATPG," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 12, pp. 1943–1956, 2013.

[12] R. Hülle, P. Fišer, J. Schmidt, and J. Borecký, "SAT-ATPG for application-oriented FPGA testing," in *15th Biennial Baltic Electronics Conference*, Oct. 2016, pp. 83–86.

[13] R. Hülle, P. Fišer, and J. Schmidt, "ZATPG: SAT-based test patterns generator with zero-aliasing in temporal compaction," *Microprocessors and Microsystems*, vol. 61, pp. 43–57, 2018.

[14] ——, "Reducing output response aliasing using Boolean optimization techniques," in *2023 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, 2023, pp. 33–38.

[15] A. Mishchenko and R. Brayton, "SAT-based complete don't-care computation for network optimization," in *Design, Automation and Test in Europe*, 2005, pp. 412–417 Vol. 1.

[16] J. Jiang, C.-C. Lee, A. Mishchenko, and C.-Y. Huang, "To SAT or not to SAT: Scalable exploration of functional dependency," *IEEE Transactions on Computers*, vol. 59, no. 04, pp. 457–467, apr 2010.

[17] B. Schmitt, A. Mishchenko, and R. Brayton, "SAT-based area recovery in structural technology mapping," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2018, pp. 586–591.

[18] M. Soeken, L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Exact synthesis of majority-inverter graphs and its applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 11, pp. 1842–1855, 2017.

[19] M. Soeken, W. Haaswijk, E. Testa, A. Mishchenko, L. G. Amarú, R. K. Brayton, and G. D. Micheli, "Practical exact synthesis," in *Design, Automation and Test in Europe*, Mar. 2018, pp. 309–314.

[20] P. Fišer, I. Háleček, and J. Schmidt, "SAT-based generation of optimum function implementations with XOR gates," in *2017 Euromicro Conference on Digital System Design (DSD)*, 2017, pp. 163–170.

[21] P. Fišer, I. Háleček, J. Schmidt, and V. Šimek, "SAT-based generation of optimum circuits with polymorphic behavior support," *Journal of Circuits, Systems and Computers*, vol. 28, no. supp01, p. 1940010, 2019.

[22] W. Haaswijk, M. Soeken, A. Mishchenko, and G. De Micheli, "SAT-based exact synthesis: Encodings, topology families, and parallelism," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 4, pp. 871–884, 2020.

[23] E. Testa, L. Amarú, M. Soeken, A. Mishchenko, P. Vuillod, P.-E. Gaillardon, and G. De Micheli, "Extending boolean methods for scalable logic synthesis," *IEEE Access*, vol. 8, pp. 226 828–226 844, 2020.

[24] H.-T. Zhang, J.-H. R. Jiang, L. Amarú, A. Mishchenko, and R. Brayton, "Deep integration of circuit simulator and SAT solver," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 877–882.

[25] B. L. Barzen, A. Reais-Parsi, E. Hung, M. Kang, A. Mishchenko, J. W. Greene, and J. Wawrzynek, "Narrowing the synthesis gap: Academic FPGA synthesis is catching up with the industry," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023, pp. 1–6.

[26] G. Audemard and L. Simon, "On the Glucose SAT solver," *International Journal on Artificial Intelligence Tools*, vol. 27, no. 01, 2018.

[27] A. Biere and M. Fleury, "Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022," in *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, T. Balyo, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2022-1. University of Helsinki, 2022, pp. 10–11.

[28] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt, "CaDiCaL 2.0," in *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, ser. Lecture Notes in Computer Science, A. Gurfinkel and V. Ganesh, Eds., vol. 14681. Springer, 2024, pp. 133–152.

[29] G. Tseitin, "On the complexity of derivation in propositional calculus," in *Automation of Reasoning*, ser. Symbolic Computation, J. Siekmann and G. Wrightson, Eds. Springer Berlin Heidelberg, 1983, pp. 466–483.

[30] R. Drechsler, "Using synthesis techniques in SAT solvers," in *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, 2004.

[31] A. Mishchenko *et al.*, "ABC: A system for sequential synthesis and verification," 2012. [Online]. Available: http://www.eecs.berkeley.edu/~alanmi/abc

[32] N. Een, A. Mishchenko, and N. Sörensson, "Applying logic synthesis for speeding up SAT," in *Theory and Applications of Satisfiability Testing – SAT 2007*, J. Marques-Silva and K. A. Sakallah, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 272–286.

[33] I. Lynce and J. P. Marques-Silva, "The interaction between simplification and search in propositional satisfiability," in *CP Workshop on Modeling and Problem Formulation*, November 2001.

[34] L. Xin, "Improving test pattern generation with implication learning," *Procedia Environmental Sciences*, vol. 11, pp. 125–131, 2011, 2011 2nd International Conference on Challenges in Environmental Science and Computer Engineering (CESCE 2011).

[35] P. Baumgartner and F. Massacci, "The taming of the (X)OR," in *Computational Logic — CL 2000*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 508–522.

[36] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "SIS: a system for sequential circuit synthesis," EECS Department, University of California, Berkeley, Tech. Rep. UCB/ERL M92/41, 1992.

[37] H. Hoos and T. Stutzle, "SATLIB: an online resource for research on SAT," in *SAT2000*. IOS Press, 2000, pp. 283–292.

[38] B. University of California, "Berkeley logic interchange format (BLIF)," 1992.

[39] B. Selman, "Stochastic search and phase transitions: AI meets physics," in *International Joint Conference on Artificial Intelligence*, vol. 1. Morgan Kaufmann Publishers Inc., 1995, pp. 998–1002.

[40] P. Fišer and J. Schmidt, "A comprehensive set of logic synthesis and optimization examples," in *12th. Int. Workshop on Boolean Problems (IWSBP)*, 2016, pp. 151–158. [Online]. Available: https://ddd.fit.cvut.cz/www/prj/Benchmarks/

[41] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "SATzilla: portfolio-based algorithm selection for SAT," *Journal of Artificial Intelligence Research*, vol. 32, p. 565–606, 07 2008.