SAT-Sweeping Enhanced for Logic Synthesis

Luca Amarú*, Felipe Marranghello*, Eleonora Testa†, Christopher Casares*, Vinicius Possani*,
Jiong Luo*, Patrick Vuillod*, Alan Mishchenko‡, Giovanni De Micheli†
*Synopsys Inc., Design Group, Sunnyvale, California, USA
†Integrated Systems Laboratory, EPFL, Lausanne, Switzerland
‡Department of EECS, University of California, Berkeley, USA

***Abstract*—SAT-sweeping is a powerful method for simplifying logic networks. It consists of merging gates that are proven equivalent (up to complementation) by running simulation and SAT solving in synergy. SAT-sweeping is used in both verification and synthesis applications within EDA. In this paper, we focus on the development of a highly efficient, synthesis-oriented, SAT-sweeping engine. We introduce a new algorithm to guide initial simulation, which strongly reduces the number of false candidates for merge, thus increasing the computational efficiency of the sweeper. We revisit the SAT-sweeping flow in light of practical considerations for synthesis, with the aim of proving all valid merges and ensuring fast execution. Experimental results confirm remarkable speedup deriving from our methodology, up to 10× for large combinational networks, and better QoR as compared to previous SAT-sweeping implementation. Embedded in a commercial synthesis flow, our proposes SAT-sweeper enables area and power savings of 1.98% and 1.81%, respectively, with neutral timing at negligible runtime overhead, over 36 testcases.**

I. Introduction

Original SAT-sweeping algorithms [1]–[5] are designed to work for both synthesis and verification, almost interchangeably. However, synthesis problems have several peculiarities that can be exploited to design higher efficiency SAT-sweeping algorithms. For example, logic networks encountered during synthesis are often shallow, mapped into gates, and contain only a few percent of mergeable gates. In the context of industrial quality logic synthesis, SAT-sweeping is asked to find all advantageous merging opportunities within limited runtime, i.e., making the best use of the given runtime budget. By analyzing the runtime profile of traditional SAT-sweepers over synthesis benchmarks, it turns out that more than 95% of the runtime is spent in proving non-equivalence, and propagating the consequences of non-equivalence, rather than proving equivalences that are useful for synthesis. Ideally, we would like SAT-sweeping to spend majority of the runtime in proving equivalent answers, thus bringing up the computational efficiency for this optimization.

In this paper, we introduce a new algorithm to guide initial simulation in SAT-sweeping, which is capable of dramatically decreasing the number of false candidates for merging. We propose enhancements to the SAT-sweeping flow, exploiting practical considerations on the type of logic networks received by the SAT-sweeper, in an industrial synthesis context. We integrate our technologies in a new SAT-sweeper engine and we compare it to a previous state-of-the-art implementation. We demonstrate up to one order of magnitude speedup for difficult industrial testcases, and 1.5× speedup on average. We also show better Quality of Results (QoR), as more merges are naturally proven by the new SAT-sweeper. Integrated in an industrial synthesis tool, the proposed SAT-sweeper enables area and power reductions of 1.98% and 1.81% , respectively, neutral timing, at the moderate runtime cost of 0.7%, measured post Place and Route (PnR) over 36 industrial testcases.

The remainder of this paper is organized as follows. Section II gives some background on SAT-sweeping. Section III studies the computational efficiency of traditional SAT-sweeping algorithms, in the context of logic synthesis, and highlights opportunities for improvements. Section IV describes a novel algorithm to guide initial simulation patterns, using fast SAT-solving, which can drastically filter false candidates for merging by accounting for many counter-examples in sweeping. Section V details our new SAT-sweeping engine, integrating the technologies presented in Sections III and IV. Section VI shows experiments for the proposed SAT- sweeper over industrial benchmarks and compares the results to state-of-the-art solutions. Section VII concludes the paper.

II. Background and Motivation

This section provides background on SAT-sweeping and discusses the motivation for this work.

A. SAT-sweeping

The advances in SAT solving technology over the past decades enabled many SAT based methods to be effective and scalable in *Electronic Design Automation* (EDA) [5]–[15]. SAT-sweeping is one such method that identifies equivalent gates (up to complementation), [1], [2]. SAT-sweeping uses SAT to check if two nodes can be merged. If the nodes cannot be merged, the SAT solver provides a counter-example, i.e., an input assignment under which the two gates can be simulated to different values. A naive implementation of SAT-sweeping would test all possible pairs of nodes. To alleviate this problem, simulation is extensively used in SAT-sweeping to reduce the number of calls to the SAT solver. By using initial random simulation, gates can be grouped into *equivalence classes*, i.e., class of gates that always simulate to the same value. Calls to SAT for proving, or disproving, equivalencies are then only needed for gates belonging to the same class. This already drastically reduces the number of SAT queries.

Fig. 1 shows a partial logic network, embedded in a larger circuit context, which we will use as example. The example is based on three different sets of simulation vectors, i.e., V_1, V_2 and V_3, which are applied at the network inputs so that corresponding simulation values are produced for all gates as

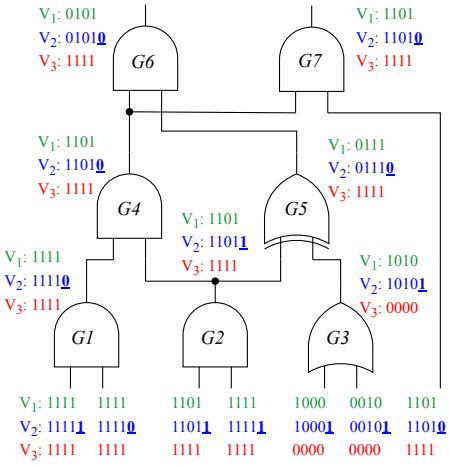

Fig. 1. Zoomed window of logic for sat-sweeping example, embedded in a larger design, with associated simulation values for equivalence classification.

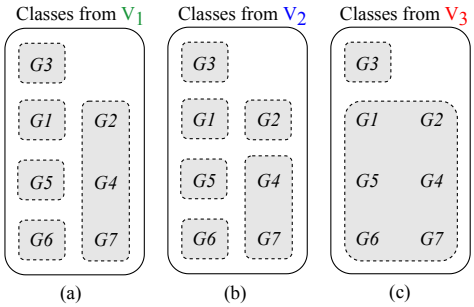

Fig. 2. Equivalence classes for the circuit in Fig. 1 obtained (a) from V_1, (b) from V_2 and (c) from V_3.

shown in Fig. 1. By applying V_1 to simulate the network in Fig. 1, we obtain the initial equivalence classes illustrated in Fig. 2(a). There is only one class with more than one gate. This class contains gates *G4*, *G2* and *G7*. Only merges among these nodes have to be tested.

Since the initial simulation is not exhaustive, gates that are neither equivalent nor complementary to each other can end up in the same class. Indeed, equivalence classes are only as good as the distinguishing power of simulation. In order to address this limitation, counter-examples provided by SAT for the non-equivalent cases can be used to incrementally re-simulate the circuit and refine equivalence classes.

In this example, gates *G2* and *G7* belong to the same equivalence class. When *G2* and *G7* are tested for equivalence the SAT solver finds an input assignment that proves that *G2* and *G7* cannot be merged. This input assignment is used as a counter-example that contributes to incrementally extend the simulation vectors from V_1 into V_2, which comprises new simulation bits highlighted (underscored) in Fig. 1. Therefore, by re-simulating the full network in Fig. 1 based on V_2, equivalence classes can be refined as illustrated in Fig. 2(b). Now only gates *G4* and *G7* are in the same equivalence class.

B. Motivation

State-of-the-art SAT-sweeping algorithms intertwine simulation and SAT solving, with intelligent engines for counter-example propagation and class refinement [4]. Some algorithms iterate over all possible pair of merges to be proven. Others iterate until further refinement of the equivalence classes is possible [4]. Further enhancement and tuning of SAT-sweeping algorithms has been summarized in [16]. In the context of synthesis, the number of valid merges is usually small, within a few percent of the total number of gates, while in the context of verification a much larger number of gates can be merged (see miters [2], [17]). Ideally, *all* valid merges need to be proved and explored for maximal QoR.

In this work we focus on synthesis applications for SAT-sweeping. As previously described, the quality of initial simulation plays a vital role in the computational efficiency of SAT-sweeping. With poor simulation, equivalence classes will be larger, and refinement via counter-examples can become computationally expensive. Unfortunately, random simulation quality/coverage becomes poor just after a few levels of logic. A trivial example for this is an n-input AND function implemented as a tree of two-input AND gates. Random initial simulation from the leaves of this tree will quickly converge to an all-0 pattern at the output.

To illustrate the impact of low quality simulation on SAT-sweeping, consider the simulation vectors in V_3 derived from the context circuit embedding the network illustrated in Fig. 1. When applying V_3 to simulate the network illustrated in Fig. 1, all patterns converge to all-0 and all-1 in a few levels of logic from the primary inputs. Therefore, this low quality simulation makes almost all gates *look identical* to the eyes of a SAT-sweeper algorithm, resulting in a poor classification as depicted in Fig. 2(c). To further refine the single large equivalence class, we solely rely on counter-examples from SAT and re-simulation.

To address these situations, we would like to have a reliable methodology for initial simulation, capable of creating high quality equivalence classes, with small or no further refinement necessary. In the ideal case, the perfect initial simulation will only leave true merges to be proven by SAT. Motivated by this reasoning, we propose several improvements to SAT-sweeping algorithms, specifically targeting synthesis applications.

III. Computational Efficiency Of SAT-Sweeping

This section presents a study on the computational efficiency of SAT-sweeping. We analyze the runtime profile of state-of-the-art algorithms running with synthesis testcases.

TABLE I
RUNTIME PROFILE FOR THREE REPRESENTATIVE BENCHMARKS

Testcase	Sim	Unsat	Sat	Time	μ
No. 1	8.38	0.13	6.23	15.96	0.81%
No. 2	5.49	0.21	2.44	9.47	2.21%
No. 3	117.91	0.84	48.23	184.18	0.45%

Table I shows the runtime profile for 3 medium size industrial benchmarks. In such circuits only a few percent of equivalent gates (up to complementation) exist, as it is typical of synthesis applications. Most of the runtime is then spent in simulating these circuits, mainly coming from counter-example propagation. The second runtime contributor comes from disproving false equivalence candidates with SAT (true answers from SAT-solving) and thus getting a counter-example to distinguish them. Only a very small percentage of runtime is spent in proving actually useful equivalences. We informally

define *computational efficiency* (μ) of SAT-sweeping as the ratio of runtime spent in proving real equivalences with the rest of runtime. Using this information, we can see that the majority of the runtime, more than 95%, is spent in computation to distinguish false candidates. This is true not only for the three testcases showed in the table above, but for many industrial testcases we experimented with, not included here for the sake of brevity. The reason for this inefficiency stands in the limitation of initial simulation. As previously mentioned, the reach of random initial simulation rapidly decreases with circuit depth.

TABLE II
NUMBER OF NON-TOGGLING GATES WITH RANDOM INPUT PATTERNS

Testcase	PI	PO	Gates	Levels	0 Toggle Rate
No. 1	9,869	26,228	21,746	41	6.92%
No. 2	22,615	37,929	124,753	46	10.25%
No. 3	43,989	81,090	119,406	42	18.20%

Eventually, some gates would have simulation patterns where no toggling at all happens: these are all-1 and all-0 simulation patterns. Table II considers the previous testcases and shows how many gates fall into this condition (zero toggle rate simulation patterns): from 6.92% to 18.20% of total number of gates. Considering a larger number of testcases, a range between 5% to 25% is typical, with outliers having up to 60% non-toggling gates under the initial simulation patterns. Outliers especially happen when the test circuit has locking logic, e.g., select logic, reset logic, etc. Non-toggling gates fall in the same equivalence class for SAT-sweeping, relying on counter-example generation and propagation to refine it. In the next Section we present a technique to alleviate this issue.

IV. SAT-GUIDANCE FOR INITIAL SIMULATION

We propose a novel technique to guide initial simulation patterns in SAT-sweeping. The goal is to narrow down the quality gap between initial and final equivalence classes, in the shortest time possible. In order to achieve this goal, we start by defining how the simulation patterns at each gate in the circuit should look like, after initial simulation. Firstly, we want to avoid gates that have all zeros or all ones in their simulation pattern. Then, we also want to avoid gates that have only a few ones and rest zeros, and viceversa. More in general, we target simulation patterns with high toggle rate[1]. On top of this, we target diverse distribution of bits in the simulation patterns with similar toggle rates. Experimentally, these characteristics of simulation patterns correlate well with high quality equivalence classes.

The constraints on the simulation patterns' characteristics can be efficiently formulated as a SAT problem. The SAT problem, if satisfiable, will produce new assignments at the primary inputs satisfying the set of constraints. For this procedure to be efficient in the context of SAT-sweeping, the calls to the SAT solver need to be resolved very quickly. A low conflict budget given to the SAT solver meets this requirement.

Alg. 1 depicts the pseudocode for our proposed SAT-based algorithm. First the gates are sorted in inverse topological order for processing. The idea is that by fixing the simulation patterns of gates at the upper levels, closer to the outputs, other gates with low toggle rate at the lower levels will be automatically handled. This makes especially sense in the context of synthesis, where circuits are relatively shallow and SAT problems can be solved with limited effort. A first round of initial random simulation is conducted with initial input patterns S. The main loop begins by considering each gate G_i, and checking if the properties of its simulation pattern respect the desired characteristics, e.g., minimum toggle rate. Note that more elaborated characteristics can be added to the check, such as distribution of bits in the pattern, alignment w.r.t. other patterns, etc. Without loss of generality, we focus on toggle rate property for Alg. 1. If the desired toggle rate is not met, then the next bit for G_i is decided, for example by flipping the last bit in the pattern. At this point, a SAT problem is formulated where the value of next bit value is assumed at the output of G_i. The number of conflicts, propagations and other metrics of the SAT solver are decided on the basis of the *pattern optimization effort* provided as input to the algorithm. In general, just a few tens of conflicts are enough to solve most of these SAT problems. Details on the circuit-level SAT formulation will be given in the next Section.

Algorithm 1 SAT based guidance of simulation patterns

Input: Network N, pattern optimization effort
Output: Optimized simulation patterns S for network N.

```
1: list ← inverse_topological_sort(N)
2: S ← initial_rand(PI)
3: for gates G_i in list do
4:     if G_i has low toggle rate then
5:         next_bit ← increase_toggles(G_i)
6:         next_PI_patt ← SAT-solving(next_bit@G_i, N)
7:         if next_PI_patt is ∅ then
8:             continue
9:         end if
10:        new_patt ← next_PI_patt ⋃ next_PI_patt(1-d)
11:        S ← S ∪ new_patt
12:        Incrementally simulate N with S
13:    end if
14: end for
15: return S
```

The SAT call in Alg. 1 can give three answers: (i) unSAT, which means that the gate G_i cannot assume the desired logic value "next_bit" under any input assignment, (ii) SAT, which means that $G_i =$ "next_bit" is possible and an input pattern for this is returned by the solver and (iii) *unDETermined (unDET)*, which means no answer was found within the solving budget. In the unSAT and unDET cases, no new *Primary Input* (PI) pattern can be added to the set of input simulation patterns S. In the SAT case, a new input pattern, also called SAT assignment, is retrieved from the SAT solver. The amount of circuit PIs necessary to guarantee a SAT assignment can be minimized by different techniques. Indeed, it is desirable to receive SAT assignments that affect the minimum number of PIs, as this corresponds to minimal (re)simulation and better compaction of learned patterns into words. Incremental SAT calls, including constraints on reducing the number of PIs,

[1]We define toggle rate as the ratio of bit-toggles over the bit-string length. For example 1001 toggles two times (1-0, 0-1), over four bits, so TR = 0.5.

can be triggered for this purpose when higher optimization efforts are specified in input to the algorithm. To continue the analysis of Alg. 1, we assume to have retrieved a valid new input pattern from the SAT solver. Instead of just adding this single new pattern, we follow the ideas presented in [4] and also add 1-distance vectors w.r.t. the new pattern, i.e., simulation vectors that have Hamming distance 1. In several practical cases, distance 1 vectors are also very likely to satisfy the same SAT problem. The number of distance 1 vectors to be added depends on the optimization effort and distance from target toggle rate. We incrementally simulate the new input pattern(s). The output of Alg. 1 is an optimized set of input patterns for simulation and an up to date simulated network. This serves as an enhanced starting point for building equivalence classes and running the core sweeping algorithm.

V. SAT-Sweeping Framework For Synthesis

This section proposes a novel SAT-sweeping framework, tailored for synthesis applications. We list hereafter some key observations, made when synthesizing industrial circuits, that are of interest to SAT-sweeping:

- Circuits in synthesis are shallow, they have in the order of tens levels of logic, up to hundreds in rare cases.
- Circuits in synthesis are not highly redundant. There are only a few percent equivalent gates to prove.
- Equivalencies in synthesis are easy to prove. This is in contrast to equivalencies in verification, that become harder with larger depth (e.g., miters).
- Processing gates top-down vs. bottom-up makes a difference. If we prove that a top gate can be removed, then we do not need to process its MFFC. Similarly, a counter-example for a top gate is likely to distinguish false candidates in its fanin cone of logic. So top-down is preferable for runtime. This is in contrast to verification where bottom-up is preferable to incrementally reduce the complexity of top equivalencies (e.g., miters).
- Depending on the synthesis step, the circuits can be mapped into gates. Gates can be decomposed into *shadow AIGs* or directly translated into CNF, following the costing proposed in [18], [19].

Based on these observations, we aim at designing a SAT-sweeper for synthesis capable of unveiling *all* valid merging opportunities with small runtime budget. Indeed, leaving merging opportunities unexplored can hurt later stages in synthesis, as the redundant logic would be placed, sized, buffered, etc. resulting in both a QoR cost and runtime cost.

Please note that the observations above hold on average, as outlier circuit cases can also be seen in synthesis. Special handling for outlier cases is necessary to ensure general scalability. For example, when a circuit has too many levels, or after getting too many unDET answers, the SAT-sweeper can be re-configured on-the-fly to operate in *"verification mode"*.

In the next two subsections we (i) discuss flow level considerations for sweeping and (ii) propose our enhanced SAT-sweeping algorithm.

A. Flow Level Considerations

At the flow level, there are various entry points in synthesis where SAT-sweeping could be called. As runtime is a main concern in modern synthesis tools, and a limited number of SAT-sweeping calls can be afforded, it is important to study where this optimization improves QoR the most in the flow.

At the HDL level, sometimes there is logic that can be shared between modules. There might also be redundancies and unseen duplications in the way operators are unrolled during high level optimization. For this reason, we run sat-sweeping first at the beginning of the synthesis flow. Getting rid of this HDL-level equivalencies early also helps the runtime of the downstream flow because fewer cells needs to optimized, placed and routed. Other than intrinsic HDL equivalencies, many other equivalent gates are created during timing optimization. When running delay restructuring, many aggressive Boolean transformations are applied, e.g., collapsing followed by Shannon's decomposition, exact delay synthesis [6], duplication of logic in the critical path, etc. At the end of the timing optimization loop, the critical paths moved across the circuit, and some of the area traded for timing during the first iterations can now be recovered. We call SAT-sweeping to merge back the duplicated logic, coming from restructuring or duplication moves, while costing the delay at each merge operation.

B. SAT-Sweeping Algorithm

The SAT-sweeping algorithm we propose is depicted by Alg. 2 and operates as follows. First, simulation patterns are derived by the SAT-guided initial pattern algorithm detailed in Alg. 1. The quality of simulation patterns is key to contain the size of equivalence classes, thus limit the number of calls to the SAT solver. With the high quality simulation patterns provided by Alg. 1, we start computing equivalence classes up to complementation. Then we sort the list of gates to be processed by sat-sweeping in inverse topological order, i.e., exploring the circuit top-down from primary outputs to primary inputs. As previously discussed, the rationale is to minimize redundant computation: for example if two outputs are equivalent, we don't want to run sat-sweeping on the cone of logic being deleted (which would instead happen in a bottom-up approach). At this point, we start processing the sorted gates. We call the current gate *candidate*, as it is a candidate gate to be deleted and replaced by a topological preceding gate. We immediately check for conditions for skipping the present candidate. For example, gates may be marked as *don't touch* or *don't try as candidate*. Many other skipping conditions include synthesis restrictions. We consider equivalence classes for positive and negative (complementary) polarity as one general class, where negative polarity gates just translate in positive polarity ones with an output inverter. The general class is then ordered topologically, but all positive polarity gates are considered foremost. This ensures the most advantageous merge is tried first, i.e., no extra inverter gate. At this point, each member of the generalized equivalence class is attempted for a merge, and is called *driver*. Conditions for skipping a driver with the current candidate are evaluated. These conditions include non-precedence in topological order, so that no loops are introduced, special markings on driver, and timing checks for timing-bounded merges. If no skipping condition applies, then the equivalence problem is translated

Algorithm 2 SAT-sweeping for synthesis

Input: Network N
Output: Optimized network N.

```
1: S ← SAT_guided_simulation_patterns(N)
2: classes ← compute_init_equiv_classes(N, S)
3: list ← inverse_topological_sort(N)
4: for each gate G_i in list do
5:     candidate = G_i;
6:     if must_skip(candidate) then
7:         continue;
8:     end if
9:     gen_class = class(G_i) ⋃ (INV+class(¬G_i));
10:    sort(gen_class) in topo order, positive pol. class first;
11:    for each gate G_j in gen_class in order do
12:        driver = G_j;
13:        if must_skip(driver, candidate) then
14:            continue;
15:        end if
16:        eq = sat-solver(candidate ⊕ driver);
17:        if eq == unDET then
18:            mark_mffc_dont_try(candidate);
19:            break;
20:        end if
21:        if eq == unSAT then
22:            Patch fanins of candidate fanouts to driver;
23:            mark_mffc_dont_try(candidate);
24:        else /* eq = SAT at this point */
25:            get counter-example from sat-solver;
26:            incremental simulation of counter-example;
27:            lazy refinement of equiv. classes;
28:        end if
29:    end for
30: end for
```

into CNF and sent for solving into the sat-solver. During the CNF translation, preference is given to the method (gate to cnf, sop to cnf, aig to cnf, lut-covering to cnf) which minimizes number of variables [19]. Details on the CNF converter and setting of the SAT-solver are omitted for the sake of brevity. If the answer provided by the solver is unDET, it means the conflict and propagation budget[2] provided to the solver were not sufficient to obtain a solution. In this case, it is likely that logic included is either not CNF friendly or just translates to a hard SAT problem (recall that CNF-SAT is NP-complete). The MFFC rooted at this candidate is also likely to lead to unDET answers. To avoid this situation, the candidate gate is marked as *don't try* together with its MFFC. This provides maximum runtime speedup and scalability. Please consider that this condition only happens rarely in synthesis applications, but we equipped our proposed algorithm to deal with it in the most scalable way. When we get unSAT answers from the solver we proceed with the merge, by reconnecting the fanins of candidate fanouts to the driver. In case of negative polarity a new INV gate is instantiated. This leaves the MFFC rooted at candidate dangling. We mark that MFFC as *don't touch*. If we instead get SAT answer, we proceed with getting the counter-example from the solver, propagating it through the simulation network and then refining the equivalence classes accordingly. Dead nodes are removed at the end of the procedure.

[2]Conflict and propagation budgets are used to limit runtime instead of absolute runtime bailouts, which are non-deterministic.

VI. Experimental Results

This section presents experimental results for our proposed sat-sweeping engine. We first apply SAT-sweeping directly on combinational logic circuits extracted from industrial designs. Then, we show results at flow level, where sat-sweeping is embedded in a commercial synthesis solution.

A. Combinational Networks SAT-sweeping Results

We extracted combinational networks from 11 industrial designs. These networks are mapped and vary in complexity from 15k gates to more than 1M gates. For the sake of comparison, we considered a previous SAT-sweeper implementation which includes state-of-the-art AIG techniques. For our novel implementation, we provide runtime results with and without the proposed SAT guidance techniques. Table III shows results for gate count (GC), runtime and network characteristics. The first column is the design name, the second and third columns are, respectively, the number of inputs and number of outputs. The fourth and fifth columns are the GC and area before sat-sweeping. Then for each sat-sweeping we show the GC and area after optimization as well as the runtime in seconds. For the new sat-sweeping columns "rt w/o guid." and "w/ guid." are the runtimes without and with the novel simulation guidance algorithm. Table IV shows details on the proof/counter-examples encountered during SAT-sweeping with and without SAT guidance. More specifically, it shows the number of total merge candidates and then the ratio of proven merges and computed counter-examples. As example, consider *design1* in Table IV. In the "w/o guid." column, we have 2895 total candidates, and proof over counter-examples ratio 0.39, implying 812 proofs and 2082 counter-examples. The set of benchmarks is diverse on purpose: there are benchmarks showing many gate merging opportunities while other benchmarks do not. We aim at making sure that our proposed methodology is highly scalable in both cases. The cases with more merges possible are expected to have higher speedup, deriving from our sat-guidance. We notice that even without sat-guidance runtime is quite better than previous implementation of the sweeper. This is because some of the algorithmic enhancements are orthogonal to initial simulation: top-down exploration, improved CNF generation from gates, marking of cones of logic where merges cannot happen, etc. It is also worth mentioning that QoR with and without sat-guidance remains the same for the benchmarks of Table III, since all merges in this set of benchmarks can be proven with the given solving budget. However, in a full design flow scenario, the speedup obtained by SAT guidance can be traded to increase solving budget and get more merges.

Considering runtime, we observe remarkable speedup for most benchmarks. For only one benchmark (*design10*) we observe virtually the same runtime because random initial simulation already gets a small number of counter-examples (few tens). For benchmarks rich in gate merging opportunities,

TABLE III
NEW SAT-SWEEPING RESULTS OVER INDUSTRIAL SYNTHESIS TESTCASES (RUNTIME IN SECONDS)

					Previous SAT-sweeper			New SAT-sweeper			
Benchmark	#In	#Out	GC	Area (bef.)	GC	Area (aft.)	runtime	GC	Area (aft.)	rt w/o guid.	w/ guid.
design1	22,615	37,929	124,753	696,421	123,929	694,946	790.6	123,830	694,762	559.4	362.0
design2	43,989	81,090	119,406	507,146	117,455	503,231	165.9	117,400	503,170	42.9	27.5
design3	6,594	10,012	15,090	68,367	14,103	65,456	89.3	14,090	65,417	28.9	15.2
design4	21,232	21,669	156,148	962,003	156,018	961,700	1,423.4	156,001	961,650	72.3	55.5
design5	114,672	140,376	170,688	592,339	169,796	589,581	229.7	169,790	589,504	26.2	19.3
design6	39,258	487,478	950,536	5,738,499	NA	NA	TO(12h)	922,092	5,664,237	2,099.0	561.7
design7	47,849	55,326	51,679	246,681	51,517	246,047	53.0	51,489	245,938	10.1	6.8
design8	47,009	2,424	140,296	186,682	47,842	94,226	100.2	47,842	94,226	61.4	60.6
design9	50,305	95,136	206,888	1,462,777	205,212	1,385,719	1,118.9	203,522	1,375,168	132.8	84.8
design10	58,439	32,670	161,296	457,580	151,176	424,158	1,821.4	151,176	424,161	10.5	10.6
design11	113,533	17,305	1,271,694	2,543,386	NA	NA	TO(12h)	1,233,985	2,365,987	41,900.9	4,163.2

TABLE IV
SAT-GUIDANCE: NUMBER OF MERGING CANDIDATES

Benchmark	# w/o guid. (p/c)	# w/ guid. (p/c)
design1	2,895 (0.39)	1,361 (1.48)
design2	4,951 (0.78)	2,957 (2.82)
design3	2,149 (0.84)	1,822 (1.17)
design4	2,900 (0.04)	781 (0.17)
design5	2,693 (0.27)	822 (2.25)
design6	59,179 (0.78)	29,220 (8.30)
design7	654 (0.35)	331 (1.07)
design8	95,924 (26.56)	95,222 (33.27)
design9	11,527 (0.22)	3,501 (1.47)
design10	6,742 (306.45)	6,724 (1680)
design11	863,291 (0.16)	132,524 (329)

we see often speedups in the order of 1.5-2×, and up to 10× for the case of *design11*. For *design11* we see the impressive runtime speedup thanks to SAT guidance, which reduces counter-examples from 740880 to just 401 (counter-example values can be extracted from total candidates and p/c ratio in Table IV), preserving the same number of merges.

B. Full Synthesis Flow SAT-sweeping Results

TABLE V
FULL DESIGN FLOW RESULTS ON 36 INDUSTRIAL DESIGN

Flow	Cmb. Area	Cmb. PW	WNS	TNS	Time
Baseline	1	1	1	1	1
New flow	**-1.98%**	**-1.81%**	-0.84%	+0.59%	+0.7%

We embedded the proposed SAT-sweeping in a commercial synthesis flow. We call SAT-sweeping both during technology independent optimization, and after timing optimization on a mapped netlist. We embed timing checks in both forms of level count and timer check for each merge. Table V shows the full flow results, post physical design. A baseline flow is run without the new sat-sweeper. The benchmarks chosen are 36 industrial designs. Our proposed flow improves combinational area by 1.98%, combinational power by 1.81%, maintaining neutral timing with only modest runtime overhead.

VII. CONCLUSIONS

This paper proposed a new SAT-sweeper engine, tailored to be efficient for logic synthesis applications. The proposed SAT-sweeper employs a novel SAT-guidance technique, which strongly reduces the number of false positive candidates for merge. Specialized algorithmic enhancements to SAT-sweeping are also proposed, especially regarding practical considerations on combinational networks encountered during synthesis. Experimental results show impressive speedup of SAT-sweeping for representative combinational networks, up to 10×, permitting more merges to be found with better runtime. Embedded in a commercial synthesis tool, the new SAT-sweeper enables 1.98% area and 1.81% power savings on top of current baseline, with only modest runtime overhead.

REFERENCES

[1] A. Mishchenko, S. Chatterjee, R. Jiang, and R. Brayton, "Fraigs: A unifying representation for logic synthesis and verification," 2005. Available at http://www.eecs.berkeley.edu/~alanmi/publications/2005/tech05_fraigs.pdf.

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

[3] Qi Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "Sat sweeping with local observability don't-cares," in *DAC*, 2006.

[4] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking," in *ICCAD*, 2006.

[5] M. Fujita, "Toward unification of synthesis and verification in topologically constrained logic design," *Proc. of the IEEE*, vol. 103, pp. 2052–2060, Nov 2015.

[6] L. Amarú, M. Soeken, P. Vuillod, J. Luo, A. Mishchenko, P.-E. Gaillardon, J. Olson, R. Brayton, and G. De Micheli, "Enabling exact delay synthesis," in *ICCAD*, 2017.

[7] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang, "Scalable don't-care-based logic optimization and resynthesis," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 4, pp. 34:1–34:23, 2011.

[8] P. Fišer, I. Háleček, and J. Schmidt, "Sat-based generation of optimum function implementations with xor gates," in *Euromicro DSD*, 2017.

[9] M. L. Case, V. N. Kravets, A. Mishchenko, and R. K. Brayton, "Merging nodes under sequential observability," in *DAC*, 2008.

[10] J. S. Zhang, A. Mishchenko, R. Brayton, and M. Chrzanowska-Jeske, "Symmetry detection for large boolean functions using circuit representation, simulation, and satisfiability," in *DAC*, 2006.

[11] B.-H. Wu, C.-J. Yang, C.-Y. R. Huang, and J.-H. R. Jiang, "A robust functional eco engine by sat proof minimization and interpolation techniques," in *ICCAD*, 2010.

[12] S. M. Plaza, K.-h. Chang, I. L. Markov, I. L. Markov, V. Bertacco, and V. Bertacco, "Node mergers in the presence of don't cares," in *ASP-DAC*, 2007.

[13] S. Krishnaswamy, H. Ren, N. Modi, and R. Puri, "Deltasyn: An efficient logic difference optimizer for eco synthesis," in *ICCAD*, 2009.

[14] M. Elbayoumi, M. S. Hsiao, and M. ElNainay, "Novel sat-based invariant-directed low-power synthesis," in *ISQED*, 2015.

[15] W. Haaswijk, A. Mishchenko, M. Soeken, and G. De Micheli, "Sat based exact synthesis using dag topology families," in *DAC*, 2018.

[16] Z. Hassan, Y. Zhang, and F. Somenzi, "A study of sweeping algorithms in the context of model checking," *Ganesh Gopalakrishnan University of Utah USA*, p. 30, 2011.

[17] D. Brand, "Verification of large synthesized designs," in *ICCAD*, 1993.

[18] M. N. Velev, "Efficient translation of boolean formulas to cnf in formal verification of microprocessors," in *ASP-DAC*, 2004.

[19] N. S. Niklas Een, Alan Mishchenko, "Applying logic synthesis for speeding up sat," *Conf. on Theory and Applications of SAT Testing*, vol. 4, Dec. 2007.