Deep Integration of Circuit Simulator and SAT Solver

He-Teng Zhang Jie-Hong R. Jiang
Department of Electrical Engineering
National Taiwan University, Taiwan
superpi152@gmail.com, jhjiang@ntu.edu.tw

Luca Amarú
Synopsys Inc., Design Group
Sunnyvale, California, USA
luca.amaru@synopsys.com

Alan Mishchenko Robert Brayton
Department of EECS
University of California, Berkeley
{alanmi, brayton}@berkeley.edu

Abstract—The paper addresses a key aspect of efficient computation in logic synthesis and formal verification, namely, the integration of a circuit simulator and a Boolean satisfiability solver. A novel way of interfacing these is proposed along with a fast preprocessing step to detect easy SAT instances and a new hybrid SAT solver, which is more robust for hardware designs than are state-of-the-art CNF-based solvers. The proposed integration enables a 10x speedup in essential computation engines widely used in industrial EDA tools, including SAT sweeping, combinational and sequential equivalence checking, and computing structural choices for technology mapping. The speedup does not lead to a loss in quality because the computed equivalences are canonical.

Index Terms—logic synthesis, simulation, Boolean satisfiability.

I. INTRODUCTION

Many application packages in logic synthesis and formal verification rely on an integration of a circuit simulator and a Boolean satisfiability (SAT) solver to process a sequence of related SAT problems formulated over a circuit representing a hardware design or a software program.

In a typical scenario, the simulator, which is faster than the SAT solver, helps the solver to avoid time-consuming satisfiable runs by falsifying easy properties using random or guided simulation. In addition, when the SAT solver produces a counter-example (CE), which the simulator could not find, the simulator uses the CE to disprove other related properties, thus saving upcoming SAT calls.

The above integration is not new [9][11]. However, it has a serious limitation. Since the number of SAT calls needed to solve a synthesis or verification problem is typically very high (thousands to millions, depending on the design size), it is very time-consuming to re-simulate the design after each satisfiable SAT call. Thus, in practice, *lazy simulation* is used: CEs produced by SAT are cached and re-simulated in a batch. However, the set of disproved properties is not updated between the batches, and the SAT solver keeps working on the remaining properties before the batched CEs are used to prune them. Thus, the solver performs redundant work, resulting in a substantial slow-down, especially for larger designs where complex logic makes functions of internal nodes hard to distinguish.

This research was supported in part by MOST 109-2224-E-002-008 of the Ministry of Science and Technology of Taiwan and by SRC Contract 2867.001.

The present paper addresses this limitation. It proposes a deeper integration of a simulator and a SAT solver, which allows for *eager simulation*: each CE is re-simulated immediately, thus disproving as many properties as possible, resulting in a dramatic reduction of the number of satisfiabile SAT calls and in a dramatic overall speedup.

To enable eager simulation without prohibitive runtime, we propose several dedicated components, which are essential for this *deep integration*: (1) a simulator, which interleaves global simulation of the design with local simulation of the properties currently being checked, (2) an application-specific SAT solver, which runs faster on circuits, compared to CNF-based solvers and available circuit-based solver, returning minimized CEs [15], thus reducing the simulation effort; (3) a novel method to disprove easy satisfiable properties often not disproved by random simulation, thus reducing the SAT solving effort.

The proposed integration can be exploited in 1) detecting and proving equivalent nodes in a logic network, known as *SAT sweeping* [11][10], 2) combinational equivalence checking [9][16]; 3) sequential signal correspondence [17], 4) computing structural choices to boost the quality of technology mapping [13][4], 5) moving net names between netlists when the names are lost during synthesis, etc.

Experimentally, the proposed integration was used in the present work to perform SAT sweeping, equivalence checking, and structural-choice computation for large hardware designs. In many cases, the runtime was reduced about 2x-10x compared to the best implementations of the corresponding operations in ABC [3].

The rest of the paper is organized as follows. Section 2 contains relevant background. Section 3 discusses the proposed integration and its components. Section 4 contains experimental results. Section 5 concludes the paper.

II. BACKGROUND

A. Boolean function

In the presentation below, *function* refers to a completely specified Boolean function $f(X)$: $B^n \rightarrow B$, $B = \{0,1\}$. The *support* of function f is the set of variables X, which influence the output value of f. The support size is denoted by $|X|$. A *minterm* of the function is an assignment of its inputs, for which the function evaluates to 1.

B. Boolean network

A *Boolean network* (or *circuit*) is a directed acyclic graph (DAG) with nodes corresponding to Boolean functions and edges corresponding to connections between the nodes.

A node *n* may have zero or more *fanins*, i.e. nodes driving *n*, and zero or more *fanouts*, i.e. nodes driven by *n*. The primary inputs (PIs) are nodes without fanins. The primary outputs (POs) are a subset of nodes of the network, connecting it to the environment. A *transitive fanin (fanout) cone* (TFI/TFO) of a node is a subset of nodes of the network, which are reachable through the fanin (fanout) edges of the node. The *TFO support* of a node is the set of primary outputs reachable through the fanouts of the node.

C. And-inverter graph

An *and-inverter graph* (AIG) [7] is a combinational Boolean network composed of two-input and-nodes and inverters. The fanins of internal and-nodes and of the POs can be inverted. The inverter is represented as a bit mark on a fanin edge rather than a separate node.

An AIG manager is a data-structure for maintaining and manipulating AIGs in a software implementation. A typical AIG manager enforces the following invariants:

- Constants are propagated (the constant 0 can only feed into a primary output).
- And-nodes are structurally hashed (each and-node has a unique pair of fanins, possibly complemented).
- The AIG nodes are stored in a topological order (that is, the node fanins precede the node itself) and referred to using a unique integer ID, giving the node's place in the order (with the constant 0 node having ID equal to 0).
- Fanins of and-nodes are ordered using their node IDs.
- There are no dangling nodes (nodes without fanout).

One way to derive an AIG for a general Boolean network, is to factor the functions of logic nodes. Then the AND/OR gates in the factored forms are converted into two-input and-nodes and inverters using DeMorgan's rule and added to the AIG in a topological order.

Due to their compactness and homogeneity, AIGs have been used as a de-facto standard for representing logic networks in various applications, including this work.

D. Boolean satisfiability

A *satisfiability problem* (SAT) is a decision problem that takes a propositional formula of a Boolean function and answers the question whether the formula is satisfiable, that is, whether the Boolean function is not a constant 0. A formula is *satisfiable* (SAT) if there is an assignment of variables called a *counter-example* (CE) such that the formula evaluates to 1. Otherwise, the formula is *unsatisfiable* (UNSAT). A software program that solves SAT problems is called a *SAT solver* [5][2].

E. Conjunctive normal form

To use the SAT solver, important aspects of the problem are encoded using Boolean *variables*. Presence or absence of a given aspect of the problem is represented by a positive or negative *literal* of the variable. A disjunction of literals is called a *clause*. A conjunction of clauses is called a conjunctive normal form (CNF). A CNF is loaded into a SAT solver and manipulated by it to determine whether the CNF is *satisfiable* or *unsatisfiable*. When the CNF is built to check the equivalence of two nodes in the circuit, a satisfiable call implies that the equivalence does not hold.

F. Circuit-based SAT solving

In logic synthesis and formal verification, which deal with hardware designs, a large portion of the CNF is often derived from a circuit representation of the design. Another option is to use *circuit-based SAT solvers*, which perform constraint propagation directly on the circuit and reserve CNF representation for application-specific constraints and learned clauses [8][9][12][19].

In a circuit-based solver, node IDs play the role of SAT variables and constraints are propagated by visiting fanins/fanouts of the nodes that have been assigned a value.

An important aspect of a circuit-based SAT solver that has no analog in CNF-based SAT solvers, is *J-frontier* [11]. Given a partial assignment of variables/nodes during SAT solving, there exist nodes whose output value is currently not implied by the values of their fanins. These nodes are called *J-nodes* because they need justification by assigning some additional fanin variables, if the partial assignment is to become a complete satisfiable assignment. The set of all J-nodes at any time during solving is called *J-frontier*.

In the case of an AIG composed of only two-input AND-nodes without XORs, the only type of a J-node is a two-input AND-node whose output has value 0 while its fanins are unassigned. In other cases, either the output value is already justified, or a new implication can be made.

The concept of a J-frontier has two important applications. It allows for the solver to stop and return "satisfiable" when the J-frontier is empty, that is, when there are no unpropagated assignments. In circuit-based applications, this typically saves time, compared to a CNF-based solver, which cannot return "satisfiable" until all variables have been assigned. Another use of the J-frontier is to make decisions during SAT solving. In this case, J-frontier-based decisions can replace or be used interchangeably with other decision-making strategies.

G. Incremental SAT solving

Modern SAT solvers, such as MiniSAT [5] and Glucose [2], offer a mode of solving, called *incremental SAT solving,* when multiple SAT calls are executed without restarting the solver. In this mode, the CNF loaded into the solver is reused with the possibility of adding new clauses between the calls. To do this efficiently, the SAT solver accepts assumptions, which are single-literal clauses holding for one SAT call.

In the case of a circuit-based solver, incrementality allows for running the solver on overlapping subcircuits without converting them into CNF. To prevent slowdowns due to excessive memory use on large designs, the solver can be restarted periodically and reloaded with only the logic cones that are needed for the remaining SAT calls.

II. CONTRIBUTIONS

We demonstrate the benefits of deep integration by developing a novel SAT sweeping engine, which detects, proves, and merges (or collects) functionally equivalent nodes in a Boolean network represented as an AIG.

SAT sweeping [11][10] is an excellent case study for this, since (1) it relies heavily on the interaction between simulation and SAT solving, (2) it produces results that are useful in many practical applications, and (3) it allows for an accurate runtime comparison between different tools because, under conditions discussed below, the set of node equivalences computed is *canonical* and thus it does not depend on a specific simulator or SAT solver used.

When it comes to practicality, SAT sweeping is a truly essential computation. When used in the context of logic synthesis, the resulting equivalent nodes can be merged, leading to area savings [1]. Alternatively, the equivalent nodes can be collected using several structurally different versions of the same netlist (for example, an area-optimized one and a delay-optimized one), leading to *structural choices*, which have been shown to improve the quality of technology mapping in both area and delay [13][4].

When used in the context of combinational or sequential equivalence checking, proving the equivalences among intermediate nodes in a topological order is key to proving the equivalences among primary outputs, especially when the proofs are hard [11]. With minor modifications, the SAT sweeping engine can be reused in various tasks, such as transferring signal names across two netlists, redundancy removal, and don't-care-based optimization [14].

The subsections below discuss the key building blocks of the proposed SAT sweeper: (1) managing equivalent nodes, (2) refining equivalences using random and guided circuit simulation, (3) a hybrid SAT solver, and (4) the high-level interaction between the components. In presenting these building blocks, we emphasize their unique contributions to the overall performance of the deep integration.

A. Equivalence class manager

Nodes of a logic network can be divided into disjoint *equivalence classes* as follows: two nodes belong to the same class if their Boolean functions are equal, up to a complement. A trivial equivalence class contains only the node itself. In practical applications, only non-trivial classes, composed of two or more nodes, are of interest. Note that internal AIG nodes whose Boolean function is a constant belong to the class of the constant 0 function.

Equivalence class computation begins by putting all nodes into the candidate equivalence class of the constant 0 node. The candidate equivalences are gradually refined, resulting in a set of proved equivalences. Nodes that could not be proved equivalent using the available resources are removed from the equivalence classes and marked as failed.

In the proposed manager, an AIG containing N nodes uses two integer arrays of size N to maintain equivalence classes. For a node, the first array stores the ID of the *representative* node, which is defined as the first node in a topological order belonging to the class. The second array stores the ID of the next node of the class, in the topological order. If the node has a unique Boolean function, its representative is the node itself and the next node is not defined.

The proposed equivalence class representation avoids memory fragmentation by allocating only two integer arrays for an AIG. It is convenient to update. For example, after a new simulation round, only those equivalence classes should be refined where a node and a representative have different simulation info. Eager refinement of equivalence classes after incremental simulation guarantees that fewer nodes are checked by the SAT solver, which saves runtime.

B. Refinement using reverse simulation

Random simulation is traditionally used to refine candidate equivalence classes [9]. Other improvements include using CEs returned by the SAT solver along with the simulation patterns that differ from CEs in one bit (so-called *distance-1 simulation patterns*) [1][8].

However, the number of satisfiable SAT calls still remains high, especially for large designs with deep logic, which tends to mask differences in node values. In order to further reduce the number of SAT calls, we developed a dedicated procedure called *reverse simulation,* which visits nodes in a reverse topological order unlike the traditional simulation, which works in a topological order.

Reverse simulation starts from two candidates, called *target nodes*, with an attempt to disprove their equivalence. For example, if A = B, it attempts to find an assignment when (A,B) = (1,0) or (0,1). The procedure traverses the TFI cone backward while assigning values at the internal nodes to be compatible with the required values at the target nodes, while marking visited nodes. The procedure terminates in two cases: (1) when the traversal is over and compatible values at the primary inputs are found, or (2) when a conflict occurs at an internal node, which tries to assign a value to a node that already has an opposite value. Similar to the traditional simulation, the reverse one can be bit-parallel, propagating many patterns in one sweep. However, our current implementation runs one bit at a time.

Reverse simulation is similar to running a SAT solver with conflict limit equal to 1. The difference is in the efficiency: the SAT solver initializes and updates many internal data-structures to make each assignment, while reverse simulation works directly on the AIG and labels the node with a two-bit flag whose value is 0 (1) if the node has value 0 (1), or 2 if the node is not visited. It can propagate multiple simulation patterns at the same time. The IDs of visited nodes are collected to clean the flag in the end. This procedure disproves equivalences with the speed that is about 20x faster than a well-tuned resource-aware integration of Glucose [2]. On average, this procedure disproves about 90% of the candidates that are not disproved by random simulation. The remaining 10% of the candidates require a robust SAT solver nevertheless.

C. A hybrid circuit-based/CNF-based SAT solver

In the past two decades, various techniques such as activity-based decisions and learned-clause minimization enabled robust CNF-based SAT solvers [5][2]. However, performance of these solvers on decision problems derived from circuits is known to be limited. Developers of circuit-based and hybrid solvers realized this and proposed various methods based on J-frontier, including justification-based decisions and backtracking [8][9][11][12][19].

Our hybrid solver is derived from Glucose [2] and has direct access to the AIG data-structure. It differs from other hybrid and circuit-based solvers in two ways. First, it uses *topological abstraction* to limit the SAT solving to only the TFI cones of the nodes whose values are being justified.

Another way to address the locality of solving would be to restart the solver more often. However, in this case, the learned clauses would be lost, which is known to slow down the SAT solving for complex logic. Thus, the topological abstraction allows our solver to grow more aggressively while limiting the scope of solving (and the fanouts visited by constraint propagation) to only the relevant logic cones.

The second unique feature of our solver is *activity-based justification* made possible by two novel data-structures, J-heap and J-watch. J-heap is similar to the traditional heap used to store variables by their activity in the CNF solvers. However, in the case of J-heap, only variables on J-frontier are stored in the heap. Also, unlike the heap in a CNF-based solver, the J-heap is empty before and after solving. J-watch allows for efficient non-chronological backtracking. A node *g* on the J-frontier is added to the J-watch list of another node *g*', if *g*' is a justification of *g*. Given the backtrack level computed by conflict analysis, J-watch lists are traversed to restore the J-frontier to the backtrack level. Other benefits of J-watch are: (1) filtering out nodes that are likely irrelevant for near-term justification; (2) keeping J-heap small to speed up push/pop operations.

Our experiments confirm that activity-based justification alone leads to an almost 2x speedup of SAT sweeping.

D. Deep integration

The SAT sweeping manager coordinates the computation. It starts by allocating the engines and computing candidate equivalence classes using random and reverse simulation, as described above. Next, the manager visits the nodes in a topological order. For example, if a node is part of a candidate equivalence class, the simulator is called to check eagerly whether the class can be refined using recently generated CEs, and if not, the SAT solver is called to check if the node is equivalent to the representative. Periodically, generated CEs are lazily re-simulated through the whole AIG to refine the remaining candidate equivalence classes.

Another role of the SAT sweeping manager is to define and enforce resource limits, such as window sizes, which allows the simulator and SAT solver to work on a small subset of nodes in a possibly large AIG.

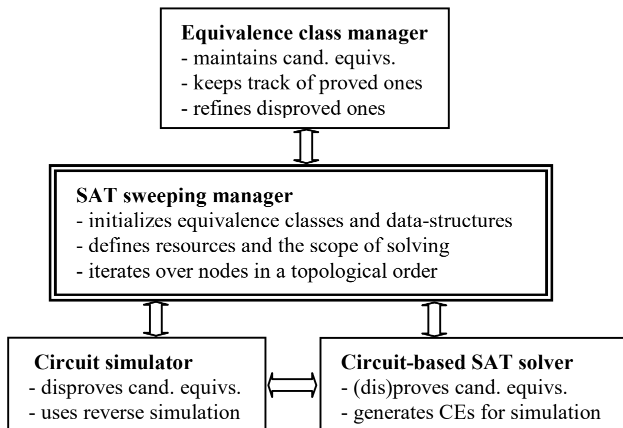

Figure 1. The high-level interaction between the components.

The following theorem states a fundamental condition that allows us to compare different SAT sweeping engines.

Theorem: Suppose an AIG is checked for equivalence of nodes without resource limits (that is, there are no timeouts and undecided SAT calls). Then, the set of proved equivalences is *canonical*, i.e., unique for any order of equivalence class refinement steps, for any simulator, and for any SAT solver (or other equivalence checking tools).

Sketch of the proof: Suppose different SAT sweepers produced two different sets of equivalent nodes belonging to the same AIG. Consider one pair of nodes that is present in one set of equivalences and absent in the other one. Since the underlying SAT solvers are used without resource limits, this node pair should have been proved equivalent by both SAT sweeper, which contradicts our assumption.

III. EXPERIMENTS

The proposed algorithms are implemented in ABC [3] and integrated into the SAT sweeping engine *&fraig* using switch "*–x*". The AIGs derived by merging the resulting equivalent nodes are verified using equivalence checker *&cec*. The experiments were run on AMD Ryzen 7 4800HS CPU with 40GB RAM. A single core and less than 1GB was used for any test case considered in this section.

The proposed SAT sweeper is compared against two similar engines, *ifraig* and *&fraig*, available in ABC. The *ifraig* engine was inspired by the pioneering work [11][10]. It handles the whole AIG in one SAT solver instance and has several other limitations. The *&fraig* engine is based on an improved AIG package and performs a resource-aware partitioning of large AIGs. To our knowledge, *&fraig* is the most efficient and scalable SAT sweeper publicly available at this time. Our engine is largely comparable to *&fraig*, except for the novel features described in this paper.

Table 1 shows the results for the first three experiments reported below in Section 4.1-4.3 using a randomly selected subset of the large benchmarks from ITC'99, ISCAS'85/89, IWLS'05 [20], and HWMCC'15 [21] benchmarks suites.

Section "Statistics" lists the benchmark name (Name) (the number in parentheses next to the name, if present, shows the timeframe count used to unfold sequential AIGs), the number of primary inputs (PI), primary outputs (PO), logic levels (Lev), and internal and-nodes in the original AIG (And) as well as in the AIG after SAT sweeping (Result). The latter number is the same for all engines because they start from the same AIG and the SAT solver conflict limit was disabled in all runs. However, the test cases running more than 900 seconds were aborted and the corresponding entries in the table contain a dash.

A. Simulation

Sections "Satisfiable SAT calls" and "Total SAT calls" in Table 1 list the number of satisfiable and total calls, respectively, performed by the solver employed in each of the three engines. The data shows that the proposed engine (column "New") dramatically reduces the number of satisfiable calls. This explains why it has a better overall runtime, since the SAT-based filtering of candidate node equivalences is typically the slowest part of SAT sweeping.

Interestingly, for two benchmarks (*s35932* unfolded for 20 timeframes and *vga_lcd*), reverse simulation disproved *all* non-equivalent node pairs. Thus, in these cases, the SAT

solver is only used to prove the equivalences, which is the "ideal" use of SAT in equivalence checking [1].

B. Circuit-based SAT solver

This experiment shows the contribution of the circuit-based features of the SAT solver introduced in this paper. Columns "NoCirc" and "New" in Section "Runtime comparison" of Table 1 show the runtime, in seconds, of the proposed SAT sweeper without and with the activity-based justification, respectively. Topological abstraction was used in both cases. Comparing the values in these columns shows that the proposed circuit-based features speed up the original CNF-based Glucose [2] almost two times when it is used for SAT sweeping.

C. Deep integration

This experiment reported in Table 1 shows that the proposed integration of simulation and SAT outperforms state-of-the-art engines by a large margin, while reducing AIG size by 7% (comparing columns "And" and "Result").

Specifically, the new engine is 10x faster than the best SAT sweeper *&fraig* and 20x faster than the earlier SAT sweeper *ifraig*, which, to our knowledge, is still used today.

In computing the averages, the timed-out benchmarks are assumed to take 900 sec (the timeout value). If a real runtime is used, the actual speedup is likely to be higher.

D. Structural choices and equivalence checking

In this experiment, we explore the performance of collecting structural choices [13][4] and combinational equivalence checking [16] when the new SAT sweeper is used in the ABC commands *&dch* and *&cec*, respectively.

We selected a subset of benchmarks from IWLS'05 [20], each containing more than 10,000 nodes and applied the following sequence of ABC commands three times: "*&st; &dch; &if –K 6; &mfs; &ps*", which is a typical script used for designs targeting FPGAs with 6-input LUTs.

Section "Statistics" lists the benchmark name (Name), the number of primary inputs (PI), primary outputs (PO), logic levels (Lev), and and-nodes in the original AIG (And).

The next three sections of Table 2 each compare the runtimes, in seconds, of before and after computations.

Section "Choice collecting runtime" compares the runtime of equivalence checking used in the first iteration of command *&dch* before and after the proposed SAT sweeping engine was integrated. The runtimes are down almost 5x, making structural choices more affordable.

Section "Script runtime" compares the runtime of the logic synthesis script, containing three iterations of *&dch* along with other commands, before and after the proposed engine was integrated. The runtimes of the complete script are reduced by about 40% (the old results are 71% slower).

Finally, Section "CEC runtime" compares the runtimes of the combinational equivalence checker *&cec* before and after the proposed engine was integrated. In this case, the runtimes are almost halved.

The results reported in Table 2 confirm that the proposed computation engine has a positive impact on the runtime of a typical logic synthesis flow, making it more affordable to the end-users, without impacting the quality of results.

IV. CONCLUSIONS

The paper explores the synergistic use of simulation and SAT solving in several key EDA applications. A novel way of deeply integrating these is proposed based on (1) a dedicated simulator, (2) an application-specific SAT solver, (3) a simulation method to disprove a large number of satisfiable properties not handled by random simulation.

The proposed implementation of SAT sweeping is found to be, on average, 20x and 10x faster than two state-of-the-art SAT sweepers, without compromising the quality. It also helps speed up other frequently used EDA computations.

Future work may include:

- Further improving the implementation by making reverse simulation bit-parallel.
- Extending the circuit-based solver to natively handle XORs, MUXes, and multi-input ANDs [19].
- Utilizing the framework to enhance other application packages, for example, don't-care-based optimization [14][18] and verification engines such as IC3 [6].

REFERENCES

[1] L. Amaru, F. Marranghello, E. Testa, Ch. Casares, V. Possani, J. Luo, P. Vuillod, A. Mishchenko, and G. De Micheli, "SAT-sweeping enhanced for logic synthesis", *Proc. DAC'20*.

[2] G. Audemard and L. Simon, SAT solver Glucose 3.0 (2013), http://www.labri.fr/perso/lsimon/glucose/

[3] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. http://www-cad.eecs.berkeley.edu/~alanmi/abc

[4] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping", *IEEE Trans. CAD*, Vol. 25(12), December 2006, pp. 2894-2903.

[5] N. Een and N. Sörensson, "An extensible SAT-solver", In *Proc. SAT'03*, LNCS 2919, pp. 502-518.

[6] A. R. Bradley, "SAT-based model checking without unrolling". *Proc. VMCAI'11*.

[7] M. Ganay and A. Kuehlmann, "On-the-fly compression of logical circuits", *Proc. IWLS'00*.

[8] M. K. Ganai, L. Zhang, P. Ashar, A. Gupta, and S. Malik, "Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver", *Proc. DAC'02*.

[9] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification", *IEEE TCAD*, Vol. 21(12), Dec 2002, pp. 1377-1394.

[10] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking," *Proc. ICCAD'04*, pp. 50-57.

[11] F. Lu, L. Wang, K. Cheng, R. Huang. "A circuit SAT solver with signal correlation guided learning". *Proc. DATE '03*, pp. 892-897.

[12] F. Lu, M. K. Iyer, G. Parthasarathy, L.-C. Wang, K.-T. Cheng, and K.C. Chen. "An efficient sequential SAT solver with improved search strategies", *Proc. DATE'05*.

[13] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, "Logic decomposition during technology mapping," *IEEE Trans. CAD*, 16(8), 1997, pp. 813-833.

[14] A. Mishchenko, J. S. Zhang, S. Sinha, J. R. Burch, R. Brayton, and M. Chrzanowska-Jeske, "Using simulation and satisfiability to compute flexibilities in Boolean networks", *IEEE Trans. CAD*, Vol. 25(5), May 2006, pp. 743-755.

[15] A. Mishchenko, N. Een, and R. Brayton, "A toolbox for counter-example analysis and optimization", In *Proc. IWLS'13*.

[16] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking", *Proc. ICCAD '06*, pp. 836-843.

[17] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal". *Proc. FMCAD'00*. LNCS, Vol. 1954, pp. 372-389.

[18] H. Riener, E. Testa, W. Haaswijk, A. Mishchenko, L. Amaru, G. De Micheli, and M. Soeken, "Scalable generic logic synthesis: One approach to rule them all", *Proc. DAC'19*.

[19] C.-A. Wu, T.-H. Lin, C.-C. Lee, and C.-Y. (Ric) Huang, “QuteSAT: A robust circuit-based SAT solver for complex circuit structure”, *Proc. DATE’07*, pp. 1313-1318.

[20] IWLS 2005 benchmarks https://iwls.org/iwls2005/benchmarks.html

[21] HWMCC 2015 benchmarks http://fmv.jku.at/hwmcc15/

TABLE 1

Comparing the number of SAT calls and the runtime of the SAT sweepers (timeout = 900 sec)

Benchmark	Statistics					Satisfiable SAT calls			Total SAT calls			Runtime comparison			
name	PI	PO	Lev	And	Result	ifraig	&fraig	New	ifraig	&fraig	New	ifraig	&fraig	NoCirc	New
b07 (100)	100	800	1457	36600	30781	265	2165	1203	5036	6937	5974	3.40	491.96	41.06	32.86
b07 (50)	50	400	732	18300	15681	97	934	448	2318	3156	2669	0.72	47.88	4.49	4.68
b18 (10)	370	230	486	817100	729122	-	-	11867	-	-	31754	-	-	671.75	146.39
b18 (15)	555	345	664	1225650	1090882	-	-	16786	-	-	46708	-	-	-	810.62
b19 (5)	120	150	474	817600	738275	-	42791	16404	-	62801	37166	-	590.99	231.69	61.37
b19 (7)	168	210	628	1144640	1029125	-	-	21239	-	-	50455	-	-	835.74	193.58
s35932(20)	700	6400	380	238960	202480	32	2197	0	5792	7957	5760	107.73	206.65	3.15	1.01
s35932(40)	1400	12800	760	477920	404960	-	-	8	-	-	11528	-	-	265.22	5.35
s38584(10)	120	2780	343	120553	112376	738	3451	2106	4141	6808	5509	13.80	20.74	11.95	4.73
s38584(15)	180	4170	513	180538	168301	1256	5318	3445	6369	10368	8558	46.99	93.15	47.26	29.10
leon2	298888	291880	58	789647	787975	461	251363	2140	1382	252239	3061	103.27	174.12	24.69	11.81
netcard	195730	97805	40	803848	802837	568	129736	6	1161	130288	599	83.57	46.14	3.82	3.68
RISC	15678	8111	40	75613	73830	1768	5461	72	2637	6310	941	14.79	1.74	0.34	0.31
vga_lcd	34247	21412	24	126708	126428	5780	17914	0	5940	18068	159	73.35	2.91	0.22	0.22
6s100	127138	97599	79	636637	627550	-	50820	2504	-	58464	10189	-	29.47	4.71	3.01
6s203b41	80192	68958	65	474322	463531	740	7110	203	6062	12301	5525	28.08	5.98	2.40	2.27
6s281b35	268334	177236	121	2076248	2058408	1533	61192	10695	7933	67489	17095	411.94	205.00	55.12	14.01
6s299b685	719410	467370	75	4111296	3809686	-	122037	902	-	179449	59972	-	103.92	20.91	19.96
6s322rb646	82513	80928	108	641468	606623	-	225761	32	-	248029	22365	-	44.36	2.88	2.22
6s342rb122	59253	56839	52	330130	319365	21110	81194	191	24140	84203	3221	545.07	18.61	0.58	0.52
6s350rb46	245680	243400	194	1550412	1545667	4939	134235	112	8255	137547	3428	790.53	58.97	8.37	7.24
6s382r	106395	104831	2752	1756654	1704409	2993	-	1493	7746	-	6246	324.68	-	36.58	47.56
6s387rb291	30615	29495	30	330186	311702	1413	4664	251	15922	18831	14760	39.93	4.61	0.92	0.81
6s392r	80920	80151	538	1599275	1583824	1653	4778	582	3948	6922	2877	133.89	48.85	1.61	1.41
Geomean				1.07	1.00	2.43	39.59	1.00	1.50	5.83	1.00	19.87	11.14	1.82	1.00

TABLE 2

Comparing the runtime of choice computation, logic synthesis, and CEC without and with the improvements

Benchmark	Statistics				Choice collecting runtime		Script runtime		CEC runtime	
name	PI	PO	Lev	And	&dch	New	Script	New	&cec	New
ac97_ctrl	4482	2251	12	14268	0.11	0.11	2.68	2.53	0.20	0.10
aes_core	1319	668	26	21522	0.40	0.25	32.78	31.98	0.82	0.39
des_perf	17850	9038	20	82650	0.94	0.65	38.21	35.63	1.44	1.13
DMA	5070	2559	27	24393	1.41	0.27	24.64	20.24	0.57	0.39
DSP	7835	3954	63	45420	4.75	0.95	77.75	64.52	2.53	1.50
ethernet	21216	10698	32	86726	6.87	0.74	56.74	35.67	1.79	0.75
leon2	298888	291880	58	789647	4586.18	71.31	15308.05	1166.47	150.50	69.66
leon3	370159	252691	59	1088122	3222.04	47.53	11745.50	1119.22	129.99	46.90
mem_ctrl	2281	1226	36	15337	0.91	0.40	8.36	6.29	0.39	0.34
netcard	195730	97805	40	803848	413.57	15.81	2252.29	523.11	46.04	26.80
pci_bridge32	6880	3533	30	22806	0.97	0.34	9.83	7.52	0.49	0.27
RISC	15678	8111	40	75613	4.27	0.91	50.84	44.45	2.69	0.84
systemcaes	1600	819	46	12384	0.16	0.16	7.19	7.55	0.28	0.22
usb_funct	3620	1858	27	15894	0.69	0.17	6.14	4.95	0.56	0.18
vga_lcd	34247	21412	24	126708	19.05	1.69	165.92	118.29	5.18	1.81
wb_conmax	2670	2189	27	47853	0.36	0.26	21.91	21.86	0.69	0.35
Geomean					4.93	1.00	1.71	1.00	1.96	1.00