Managing Don't Cares in Boolean Satisfiability

Sean Safarpour[1] Andreas Veneris[1,2] Rolf Drechsler[3] Joanne Lee[1]

Abstract

Advances in Boolean satisfiability solvers have popularized their use in many of today's CAD VLSI challenges. Existing satisfiability solvers operate on a circuit representation that does not capture all of the structural circuit characteristics and properties. This work proposes algorithms that take into account the circuit don't care conditions thus enhancing the performance of these tools. Don't care sets are addressed in this work both statically and dynamically to reduce the search space and guide the decision making process. Experiments demonstrate performance gains.

1 Introduction

Advances in Boolean satisfiability (SAT) solvers [1] [2] [6] [9] [10] [13] have made them attractive engines for solving problems in the digital VLSI design cycle [2] [8] [14] [15]. Most modern SAT solvers are based on branch-and-bound search algorithms enhanced with speed up reasoning mechanisms and robust data structures. However, in most practical VLSI cases, the circuit is "translated" into a Conjuctive Normal Form (CNF) before being processed by the tool. This process fails to keep the circuit's *structural information* which proves to be helpful in other branch-and-bound methods such as ATPG [3] [4]. For this reason, researchers investigate methods for which SAT solvers consider structural circuit characteristics [1] [2] [6] [9] to improve performance.

To address these issues, we describe a new method that allows existing SAT solvers to handle *sets of don't care conditions* derived from the circuit. Sets of don't care conditions are of great value because they provide various degrees of freedom in logic synthesis, testing and verification [3] [4]. In our context, don't cares allow the SAT solver to prune the search space. In this sense, we do not propose a new SAT solver. Rather, we describe techniques to enhance existing tools to take advantage of circuit properties and improve performance.

The proposed work allows SAT solvers to take advantage of these sets both *statically* and *dynamically*. In the first case, the CNF formula is enriched with clauses that encode the circuit's don't cares to indicate areas of the search space with no solution. These clauses are computed during a preprocessing step to enrich the CNF of the circuit. Don't cares are also handled dynamically by biasing the decision making of the solver. In this case, the solver uses circuit information "on the fly" to ignore variables (or circuit lines) that are not relevant to the goal under consideration. Experiments show significant speed-ups to a state-of-the-art SAT solver when presented with the set of circuit don't care conditions.

This paper is structured as follows. The next section contains terminology. Section 3 defines sets of circuit don't care in the context of Boolean satisfiability and Section 4 describes algorithms to manage these don't cares. Section 5 contains experiments and Section 6 concludes this work.

2 Preliminaries

We consider combinational circuits with `AND`, `OR`, `NOT`, `NAND`, `NOR`, `XOR` and `XNOR` gates. These circuits are translated into CNF using the procedure from [8]. CNF forms are expressed as a logical `AND` $(\cdot)$ of *clauses*, each of which is the `OR` $(+)$ of one or more literals. A literal is an instance of a variable x or its negation x'. We use the same letter to refer both to the circuit line and the respective line variable in the CNF, unless otherwise specified. Given a CNF formula, a SAT solver finds a variable assignment that satisfies the formula (SAT case) or it proves that the formula cannot be satisfied (UNSAT case). In the remaining paper, we assume that the reader is familiar with the terminology from [10] [13].

Line l *dominates* line l' if all paths from l' to any primary output go through l. As a base case, a line dominates itself. Structural dominators are found in linear time with algorithms such as the ones in [5]. For example, in Fig. 1, line b dominates all the circuitry shown in the dotted area including the gates that fan in to lines v and w. We call this dotted area the *dominating region* of b. The thick incoming arrows in this figure are circuit lines with one endpoint at a gate(s) with an output in the dominated region and the other endpoints at a gate(s) elsewhere. We call all these lines (arrows) the *support* of the dominated region.

To utilize certain types of don't care conditions, we partition the circuit into *fan-out free* circuits referred to as *cones*. The output of a cone is a primary output or a stem line while the input of the cone are primary input or branches (that is, output lines from other cones). Fig. 1 contains such a cone with lines $\{l, k, f, e, d, g_2, c_1, c_2, h\}$, inputs $\{f, g_2, c_1, c_2, h\}$ and output line l. Finally, the controlling value of an `AND` and a `NAND` (`OR` and `NOR`) gate is 0 (1).

[1]University of Toronto, Department of Electrical and Computer Engineering, Toronto, ON M5S 3G4 ({ ssafarpo, veneris, leejoa }@eecg.toronto.edu)

[2]University of Toronto, Department of Computer Science

[3]University of Bremen, Institute of Computer Science, 28359 Bremen (drechsle@informatik.uni-bremen.de)

3 Circuit Don't Cares in SAT

The core of the proposed technique is the efficient management of don't cares. In this Section, we first define the don't care conditions addressed in this paper and provide the motivation. Our definitions are different to the ones commonly used in literature [3] [4], i.e. they have been modified in ways to reflect *structural circuit properties*. These properties are examined during the search process of the SAT solver.

Sets of circuit don't care conditions are classified as *controllability* (or *satisfiability*) (CDC) and *observability* (ODC) don't care conditions. In particular, we address sets of don't care (DCs) conditions defined as follows:

Observability DCs: These are lines where value assignments do not influence the outcome of the current SAT solver goal. For example, when the SAT solver explores a portion of the design space where $d = 0$ (Fig. 1), value assignments for lines $\{k, e, f, g_2, c_1\}$ do not influence the outcome of the search process. These lines may assume values that are *implied* from other circuit lines (such as stems g and c) but the SAT solver does not need to *explicitly* branch on them.

Controllability DCs: These are line values that cannot be justified in the circuit under any primary input logic value assignment. In Fig. 1, combination of logic values $(e = 0, d = 0)$ is illegal under *any* primary input assignment and it belongs to the CDC space of the design. The SAT solver can benefit by discovering it early to avoid unnecessary branching on these values and thus backtracking later on.

Performance improvement using the above don't cares is achieved in the following two ways by the solver:

1. reduce the number of branches (and thus backtracks) for variables that do not influence the current goal, and
2. pre-process the circuit to introduce extra clauses in the CNF to prune the search space

In the section that follows we explain how observability don't cares materialize the first objective. Controllability don't cares (also in Section 4) are utilized for the second objective.

4 Managing Don't Cares

4.1 Observability DCs

ODCs are managed dynamically as the solver examines variable assignments and it consults the circuit structure to determine variables (circuit lines) that no longer influence the outcome of the current goal during decision making.

Consider the circuit in Fig. 1 and let $a = 0$. Following this assignment, *any* assignment on b and lines in its dominating region are not observable at any primary output. Therefore, the SAT solver does not need to consider (branch) these variables when the solution space is restricted to $a = 0$ since any value assignment on them has no impact on the final outcome of the goal.

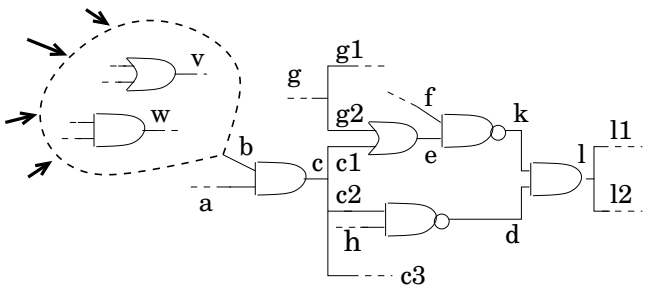

Figure 1: Observability Don't Cares

The overall procedure for ODCs is a generalization of the one described above. Let l be the variable under consideration at the current decision level. Let line l also be an input to gate G in the circuit. If the solver assigns l a value which is controlling for G then all other lines that are an input to G as well as their dominated regions are marked as *lazy*. A stem is declared lazy when all of its branches are marked lazy. In the circuit in Fig. 1, stem g (and the circuitry dominated by g) is marked lazy when both of its branches g_1 and g_2 become lazy, possibly from other distinct variable assignments at different decision levels. Variables not marked as lazy, are called *free* variables.

It is clear that the sets of lazy and free variables change as different decisions/backtracks explore various parts of the search space. Depending on the decision made, a variable can be marked lazy and unmarked later if the solver backtracks from this decision. Lazy variables are important because the SAT solver does not need to branch on them in future decision making steps. Under this operational framework, the SAT solver always selects an unassigned free variable to branch and enters the next decision level.

A similar idea where value assignments are used to exclude dominating regions from the decision process is proposed in [2]. However, the work in [2] uses these assignments to exclude clauses from the CNF unlike this work that marks those variables as lazy. In more detail, the computational gain of this proposed method comes from the following observations:

- The solver declares SAT if it reaches a decision level where no free variables are left unassigned. This may occur even if some lazy variables are unassigned (don't care). The satisfying assignment consists of value assignments to all free variables at the point where SAT is declared.
- When the solver declares UNSAT exclusion of lazy variables may produce conflicts earlier in the decision tree. Backtracks, in this case, may occur early as the solver ignores lazy variables by unnecessary branching on them.

It should be noted, a variable may be marked lazy even if it has already been assigned a value at an earlier decision level. Given a SAT outcome for a problem instance, the final value of a lazy variable may be *different* from the one assigned by the solver. To see this, the final value of the lazy variable is *implied* by value assignments of free variables in the SAT solution. In other words, the values are found by simulating

the value assignment on the free variables. Therefore, logic assignments on lazy variables for a SAT outcome can be ignored. Note that if input signal are marked lazy, these lines can be assigned X or implied values as in ATPG.

Example 1 *Recall that line b and its dominator region (in dotted circle) in Fig. 1 is marked lazy when $a = 0$. Assume that at this decision level, the CNF can be satisfied and that all lines in the support of the dominated region remain free*[2]*. Given the satisfying assignment (hypothesis), the values for line b and all lines dominated by b, such as v and w, can be determined from the set of logic values $\{0, 1, X\}$ with logic simulation of value assignments in the support (thick arrows) of the region.*

A different way to view the impact of lazy variables in the decision making process is to consider the *expanded truth table* of the family of circuits with k lines, including n primary inputs $x_1, x_2, \ldots, x_n$ and m primary outputs $y_1, y_2, \ldots, y_m$. This table is shown in Fig. 2(a) and it has 2^k rows as it contains all possible value combinations for k variables. Note that our definition and use of the expanded truth table is different to the one presented in [11].

Given a *specific* circuit in the family of circuits with k lines, one may shuffle the rows of the expanded table as follows. The top part of the table contains all 2^n valid combinations of circuit line values for the complete input vector space. Intuitively, each row in the legal part of the table can be seen as a snapshot of the circuit when simulated for the respective input vector. The bottom part of the table contains the remaining rows, that is, invalid simulation snapshots of the circuit under consideration. Some assignments in the valid space of the circuit lead to SAT while any other part of the valid space gives UNSAT depending on the goal at hand. However, the complete 2^k space of the expanded truth table can be intuitively seen as the solution space for any SAT solver.

When variables, such as v and w in Fig. 1, are marked lazy, the SAT solver excludes them from the decision process. In terms of the expanded truth table, it means that the columns corresponding to these variables can be excluded leading to a smaller table. This "compacted" truth table contains less variables for the solver to branch, as shown in Fig. 2(b), speeding up the overall performance. It should be noted that the expanded truth table is not implemented as part of the solver but we merely use it here to illustrate various concepts.

Marking variables lazy has the additional benefit of undoing some "bad" decisions made earlier by the solver. To illustrate this point, assume that the solver assigned $w = 1$ and $v = 1$ before it assigns $a = 0$ and marks w and v as lazy in Fig. 1. Also assume that there exist an indirect implication [7] $w = 1 \Rightarrow v = 0$ in the circuit. In other words, a traditional solver would *eventually* conflict and backtrack from decision $a = 0$ just to realize that $w = 1$ and $v = 1$ is an illegal combination and learn the implication $w = 1 \Rightarrow v = 0$.

Clearly, the value assignments on w and v lead the solver to search part of the invalid space of the design. However, in the suggested configuration, when a gets assigned to 0 the invalid decision on w and v is erased. This is because both

[2]If some of these lines are eventually marked lazy, a similar argument can be made using the new support of those lines.

$x_1 \ldots x_n$	w	g	a	...	v	h	$y_1 \ldots y_m$	
0 ... 0	0	1	1	...	0	0	0 ... 0	VALID
1 ... 1	1	1	1	...	1	0	1 ... 1	
1 ... 0	1	0	1	...	0	0	0 ... 0	INVALID
1 ... 0	1	1	1	...	0	0	1 ... 0	

(a)

$x_1 \ldots x_n$	w	g	a	...	v	h	$y_1 \ldots y_m$	
0 ... 0		1	1	...		0	0 ... 0	VALID
1 ... 1		1	1	...		0	1 ... 1	
1 ... 0		0	1	...		0	0 ... 0	INVALID
1 ... 0		1	1	...		0	1 ... 0	

(b)

Figure 2: Expanded Truth Table

variables are marked as lazy and the solver is presented with a new solution space, indicated by the table in Fig. 2(b), where the variables that pertain to the implication do not even exist. However, implications of lazy variables on free variables may still cause the solver to backtrack.

We may conclude, that ODCs are useful because they reduce the search space dynamically. Experiments presented later in this paper confirm this theoretical behavior. They also indicate that the proposed method reduces the number of backtracks to improve performance.

4.2 Controllability DCs

Controllability DCs are variable (line) assignments that cannot be justified under any primary input test vector. For instance, if implication $w = 1 \Rightarrow v = 0$ holds for the circuit in Fig. 1, then value assignment $(w = 1, v = 1)$ cannot be justified. Intuitively, CDCs are combinations of logic values that belong in the invalid portion of the expanded truth table of the circuit in Fig. 2(a).

Obviously, this definition of CDCs results in numerous combinations to be examined. In the proposed approach, we consider CDCs at input of cones that partition the circuit. As explained earlier, stems (cone input) present an increased complexity in digital VLSI due to their reconvergent property[3] [4]. In this work, cones and their respective stems allow for a systematic view of CDCs; however, one may consider alternate ways to handle CDCs.

CDCs are computed in terms of clauses for each cone independently in a pre-processing step. They are later added to the CNF of the circuit under consideration. The pre-processing step works as follows.

Given a cone with p inputs, pre-processing first extracts all circuitry from the original circuit that fans-in to this cone and create a new circuit with p primary outputs. For example, in Fig. 3 the extracted circuitry of cone B is shown within thick dotted lines. Following this extraction process, we run a SAT solver on this new circuitry to identify combination of values on these p lines that cannot occur.

To reduce the number of combinations needed to be verified, parallel logic simulation [4] is performed for a small number of test vectors (usually less than 700 vectors). Combination of logic values from simulation on the p inputs of the cone are accumulated since they are guaranteed not to be CDCs. We call these *viable* combinations. Viable combinations are inserted as clauses in the CNF of the logic cone where the literals take the inverted polarity of the simulation value. This way the SAT solver is left with the space of *potential CDCs* to prove. For example, if simulation of some vector gives $(b, c, d) = (1, 0, 1)$ for a cone with inputs b, c and d, then clause $(b' + c + d')$ is inserted. For cones with a large number of inputs (more than 7), the number of viable combinations returned by simulation may be large. In this case, we compress all viable combinations using BDDs [3] into cubes. This has the desirable effect to minimize the number of clauses inserted in the CNF.

Given the new CNF for the cone, the SAT solver is presented with a search space that represents a set of potential CDCs. Every SAT solution from this space identifies formally a viable combination. To reuse the computation, the solver does not reset but every new viable combination is added "on the fly" as a learned clause to force the solver to backtrack and explore the remaining portion of the solution space. Eventually, the solver will exhaust the solution space and return UNSAT.

At that point, the set of CDCs for the particular cone is known and they are later appended in the CNF of the original circuit. It should be noted, CDCs are computed only once for the circuit using the above pre-processing step but they can be used many times since, in the digital VLSI cycle, the same circuit usually undergoes multiple steps such as synthesis, testing, optimization, etc.

In some cases, CDCs may be time consuming to prove. In these cases, we limit the size of circuitry extracted. Instead of extracting logic up to the primary input of the circuit, we specify the number of *cone levels* to extract. For example, when examining CDCs for cone D in Fig. 3, we may extract circuitry that does not go past cone A or, equivalently, 2 cones of logic prior to D.

Clearly, if the solver proves a CDC using a set of cones that do not reach the primary input, this combination of values is not viable through the primary input as well. However, the opposite statement is not true and potential CDCs can be viable combinations using a small cone circuitry but they may not be viable using the primary input. Therefore, this heuristic acts as a trade-off between the amount of CDCs one proves and the time allocated to prove them.

Finally, cones are examined moving from primary input towards primary output. With respect to Fig. 3, this implies

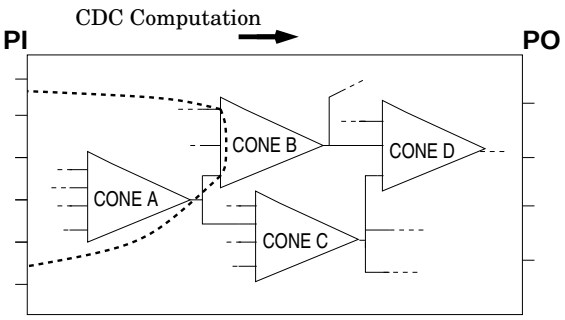

Figure 3: Controlability Don't Cares

that CDCs for cone A are examined first, cones B and C next, cone D, and so on. All CDCs proven by the tool are inserted in the CNF of the circuitry for cones examined later. This topological visit of the circuit and re-use of information has been shown helpful when dealing with circuit-based SAT instances [9].

5 Experiments

To evaluate the techniques described in Section 4, we implemented them into the latest version of the SAT solver zChaff [10]. Experiments are performed on a SunBlade 100 workstation with 512MB of memory using benchmarks from the ISCAS'85 family of circuits. The results are reported in the following tables and expanded upon in the next paragraphs. All CPU run-times are in seconds.

The performance of the enhanced solver is tested for both UNSAT and SAT cases. UNSAT experiments are performed in terms of logic verification: a circuit is either verified against a duplicate copy of itself or against an optimized version of itself using SIS [12] (`script.rugged`). Experiments of the latter category have a letter "o" following the circuit name. In the SAT cases we test the performance of the proposed techniques on some hard to detect stuck-at faults.

Performance gains for both cases are demonstrated against the tool operating with no knowledge of don't cares. For the UNSAT cases, we evaluate the performance of the ODC and CDC techniques both separately and together. In the tables that follow, we do not include small benchmarks because the original tool is fast enough (*i.e.,* under one second) and does not justify the overhead of the proposed approach.

Table 1 shows a snapshot of the CDC performance on a set of benchmark circuits at a maximum user-specified cone level of 8. Since the main contribution of CDCs is to prune the search space by introducing new clauses, we indicate how many potential CDCs are verified to be CDCs at pre-processing in column 2. The third column shows the number of potential CDCs still unproven at this cone level. The total number of CDCs tested is the sum of these two columns.

The final three columns of the table show the CPU times required for zChaff, the pre-processing time (that is, the time required for vector simulation and verification of potential CDCs), and the final solve time using the enhanced formula with CDCs. The sum of columns 5 and 6 is the total time required to solve the UNSAT problem. In the experiments it

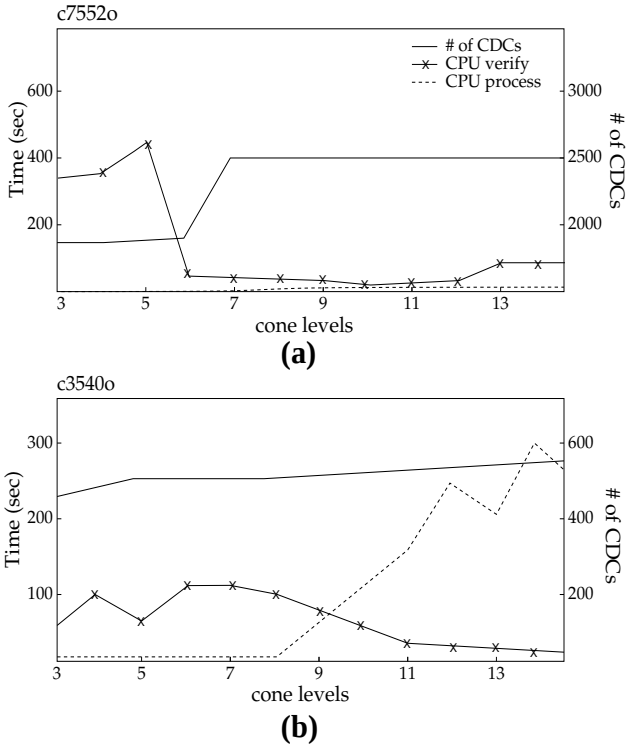

Figure 4: Controllability DCs Statistics

ckt name	CDCs proved	CDCs left	CPU (sec)		
			zChaff	process	proposed
c1908	122	0	1.4	<0.1	2.7
c1908o	85	7	2.5	<0.1	3.9
c2670	182	2	1.4	1.0	0.3
c2670o	208	3	7.2	1.3	0.2
c3540	618	14	48.9	30.0	34.7
c3540o	511	24	158.4	8.4	87.8
c5315	1170	30	69.2	1.4	59.4
c5315o	876	30	153.2	1.3	74.5
c7552	3150	34	177.1	6.6	49.3
c7552o	2501	35	282.0	5.7	97.9
Avg	942	18	90.1	5.6	41.1

Table 1: Controllability DCs in UNSAT

ckt name	zChaff		Proposed	
	back-tracks	CPU (sec)	back-tracks	CPU (sec)
c1908	5349	1.4	6953	1.9
c1908o	6930	2.5	4807	2.1
c2670	4188	1.4	3307	1.5
c2670o	11371	7.2	4223	3.3
c3540	37396	48.9	37986	69.4
c3540o	80838	158.4	44125	102.3
c5315	60989	69.2	22235	22.3
c5315o	99788	153.2	32275	36.6
c7552	110950	177.1	33909	52.7
c7552o	151062	282.0	38516	72.9
Avg	56886	90.1	22834	36.5

Table 2: Observability DCs in UNSAT

was noticed that the most benefit occurred when only CDC clauses with 5 or less literals were collected. We restrict the number of CDC clauses added because too many may degrade performance and provide little information. The numbers in Table 1 indicate improvement in performance when using CDCs.

For two circuits we study the CDC behavior in more detail. Fig. 4 illustrates the general trend of a CDC incorporated solver for circuits *c7550o* and *c3540o*. It shows the time to pre-process the CDCs, the number of proven CDCs, and the time to verify the CDC enriched circuits versus the level of cones. It is observed that a higher number of cone levels increases the number of CDCs proved. Ideally all CDCs are computed if the extracted region contains all circuitry including primary input(s). The processing time tends to increase with the number of cone levels.

The next few paragraphs elaborate on the performance of ODCs. In addition to the theory developed in subsection 4.1, we bias the VSIDS [10] variable selection process further as follows. In the original implementation, the unassigned literal that occurs most frequently is always selected for branching. In the modified version, we consider a *range* of the top 7 such literals and select the one that produces the largest amount of lazy variables. We also experimented with other heuristics that bias the variable selection process such as setting variables to the controlling value of the gate they fan-in first. However, those heuristics seem to degrade tool performance when compared to the original/proposed selection process.

Table 2 evaluates the ODC technique for the UNSAT cases. The number of backtracks incurred by the solver is often used as an indicator of its relative efficiency [13]. These numbers are indicated in columns 2 and 4 for the original tool and the one enhanced with ODCs, respectively. It is clear that the number of backtracks is usually reduced by a factor of 2 for the ISCAS'85 circuits. This is also the trend for the number of decisions made but we omit the result due to lack of space. The total solve times are presented in columns 3 and 5. We observe that the total time is reduced proportionally to the number of backtracks. This confirms the observation in [13] that computation and backtracks are related. Clearly, ODCs have more impact on large circuits where more variables are marked lazy to reduce the number of backtracks/decisions made.

The combined CDC and ODC results for UNSAT cases are presented in Table 3. The number of backtracks and the number of proven CDCs are given in columns 2 and 3, respectively. For each circuit we use a fixed cone CDC level of 7 in the pre-processing step. Process CDC times and verification times for the proposed approach are indicated in the last two columns. With the exception of *c*1908, a relatively small circuit where the additional overhead does not compensate for the improved performance, the data in Table 3 illustrate that the combined ODC and CDC methods provide significant performance gain.

SAT cases are created by adding stuck-at faults. Since most faults are relatively easy to detect (especially in small

benchmarks), both versions of the tool return a vector very quickly (under one second). Therefore, we isolate circuits and faults that dominate the test generation time and use the dynamic ODC technique to evaluate the performance for "hard" SAT cases. Results are presented in Table 4.

The number of variables that are marked lazy and are unassigned by the time the solver returns a solution are illustrated in Column 2. In terms of ATPG, these are similar to the circuit lines assigned to a logic unknown X when a vector is found. Note that in our case, the true value can be one of $0, 1$ or X. The next column has the total number of variables in the circuit. It is interesting to notice that a large fraction of the variables remain unassigned as in ATPG. Columns 4 and 5 compare the average CPU times per circuit for the original and new tool, respectively. The benefit of the ODC technique is apparent as it provides a considerable improvement for SAT cases.

6 Conclusions

Boolean satisfiability models various digital VLSI problems. For this reason, efficient SAT solvers are of interest in the research community as well as the industry. In this paper, we presented a method where existing solvers use structural circuit information to take advantage of a design's don't care space. Experiments demonstrated that these techniques substantially improve the performance of an existing state-of-the-art SAT solver.

ckt name	back-tracks	# of CDCs	CPU (sec)		
			zChaff	process	proposed
c1908	14081	122	1.4	<0.1	3.7
c1908o	6479	85	2.5	<0.1	4.1
c2670	474	182	1.4	0.9	0.4
c2670o	638	208	7.2	2.8	0.7
c3540	24613	614	48.9	26.3	34.8
c3540o	34425	507	158.4	0.9	71.7
c5315	8841	1162	69.2	1.7	9.5
c5315o	15875	868	153.2	1.7	15.5
c7552	28688	3150	177.1	8.5	44.8
c7552o	34903	2501	282.0	7.6	74.5
Avg	16902	940	90.1	5.1	26.4

Table 3: Combined results on UNSAT

ckt name	# lazy variables	# of total variables	CPU (sec)	
			zChaff	proposed
c3540	619	2092	25.7	1.4
c3540o	886	2796	77.6	39.5
c5315	1551	3953	34.8	1.5
c5315o	1679	4681	92.3	4.7
c7552	1590	5399	57.1	1.3
c7552o	1830	6578	67.7	1.3
Avg	1359	4250	59.2	8.3

Table 4: Results on SAT

References

[1] M. K. Ganai, L. Zhang, P. Ashar, A. Gupta and S. Malik, "Combinational strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver," in *IEEE DAC, pp. 747-750,* 2002.

[2] A. Gupta, A. Gupta, Z. Yang and P. Ashar, "Dynamic Detection and Removal of Inactive Clauses in SAT with Application to Image Computation," in *IEEE DAC, pp. 536-541,* 2001.

[3] G. Hachtel and F. Somenzi, "Logic Synthesis and Verification Algorithms," *Kluwer Academic Publishers,* 1996.

[4] N. Jha and S. Gupta, *Testing of Digital Systems*, Cambridge University Press, 2003.

[5] T. Kirkland and M. R. Mercer, "A Topological Search Algorithm for ATPG," in *IEEE DAC, pp. 502-508,* 1987.

[6] A. Kuehlmann, M. Ganai, and V. Paruthi, "Circuit-based Boolean Reasoning," in *IEEE DAC, pp. 232-237,* 2001.

[7] W. Kunz and D. K. Pradhan, "Recursive Learning: A New Implication Technique for Efficient Solutions to CAD Problems–Test, Verification, and Optimization," in *IEEE Trans. on Computer-Aided Design, vol. 13, no. 9, pp. 1143-1158* September 1994.

[8] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," in *IEEE Trans. on CAD, vol. 11, no. 1, pp. 4-15,* Jan. 1992.

[9] F. Lu, L.-C. Wang, K.-T. Cheng and R. Y.-Y. Huang, "A Circuit SAT Solver with Signal Correlation Guided Learning," in *Proc. of IEEE DATE, pp. 892-897,* 2003.

[10] M.H. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *IEEE DAC, pp. 530-535,* 2001.

[11] S. M. Reddy, "Complete Test Sets for Logic Functions," in *IEEE Trans. on Computers, vol. C-22, no. 11, pp. 1016-1020,* Nov. 1973.

[12] E. Sentovich, K. Singh, C. Moon, H. Savoj, R. Brayton, and A. Sangiovanni-Vincentelli, "Sequential Circuit Design Using Synthesis and Optimization," in *IEEE ICCD, pp. 328-333,* 1992.

[13] J. P. M.-Silva and K. A. Sakallah, "GRASP – A Search Algorithm for Propositional Satisfiability," in *IEEE Trans. on Computers, vol. 48, no. 5, pp. 506-521,* May 1999.

[14] P. Tafertshofer, A. Ganz and M. Henftling, "A SAT-Based Implication Engine for Efficient ATPG, Equivalence Checking and Optimization of Netlists," in *IEEE ICCAD, pp. 648-657,* 1997

[15] A. Smith, A. Veneris and A. Viglas, "Design Diagnosis Using Boolean Satisfiability," in *IEEE ASP-DAC*, 2004.