Design Diagnosis Using Boolean Satisfiability *

Alexander Smith
University of Toronto
Dept ECE
Toronto, ON M5S 3G4
smith@eecg.toronto.edu

Andreas Veneris
University of Toronto
Dept ECE and CS
Toronto, ON M5S 3G4
veneris@eecg.toronto.edu

Anastasios Viglas
University of Toronto
Dept CS
Toronto, ON M5S 3G4
aviglas@cs.toronto.edu

Abstract

Recent advances in Boolean satisfiability have made it an attractive engine for solving many digital VLSI design problems such as verification, model checking, optimization and test generation. Fault diagnosis and logic debugging have not been addressed by existing satisfiability-based solutions. This paper attempts to bridge this gap by proposing a satisfiability-based solution to these problems. The proposed formulation is intuitive and easy to implement. It shows that satisfiability captures significant problem characteristics and it offers different trade-offs. It also provides new opportunities for satisfiability-based diagnosis tools and diagnosis-specific satisfiability algorithms. Theory and experiments validate the claims and demonstrate its potential.

1 Introduction

The digital VLSI design cycle commonly starts with a behavioral description (specification) coded in some Hardware Description Language (HDL) (Fig. 1). This specification is translated to a Register-Transfer Level (RTL) description and synthesized into a gate-level (logic) implementation. Design validation and optimization steps guarantee the correctness of the product from design errors as well as its performance according to the specification. Subsequent steps involve placement, routing and physical optimization before the chip is fabricated. Diagnosis and failure analysis are the last steps before it is shipped to the customer. If the fabricated chip fails testing, it undergoes failure analysis.

Recent years have seen an increased use of Boolean Satisfiability (SAT) based tools in the design cycle. Design verification and model checking [3] [4] [8] [10], test generation [7], optimization [12] and physical design [14], among others, have been successfully tackled with

---

SAT-based solutions. This is due to recent advances in SAT solvers [9] [11] that make them efficient platforms to solve these problems. It has also been reported [10] that problems with SAT based formulations for industrial circuits are usually solved in polynomial time. This is favorable because they are intractable and the worst-case behavior of a complete algorithm today is exponential.

Although SAT-based solutions have tackled many circuit design problems, *design diagnosis* has not yet been addressed in existing literature. Given an erroneous design, a specification and a set of input test vectors, diagnosis identifies malfunctioning portions of the design. Diagnosis is integral to failure analysis in helping to improve the design cycle, increase manufacture yield and shorten the time-to-market window [2] [6].

Depending on the stage of the design cycle, shown in Fig. 1, and the type of malfunction ("soft" or "hard"), diagnosis is required during design error diagnosis (logic debugging) and during fault diagnosis. *Design error diagnosis* occurs in early stages of the design cycle where the specification is some HDL (or RTL) description and the design is a logic netlist. Malfunctions are caused by specification changes, bugs in automated tools or the human factor [1]. Logic debugging identifies lines and corrections in the erroneous netlist that correct it according to a specification. *Fault diagnosis* occurs when a fabricated chip fails testing. Given a faulty chip and a netlist, fault diagnosis identifies locations in the correct netlist by injecting faults into it until the netlist emulates the behavior of the faulty chip. Since both problems have similar goals, we describe this work in terms of fault diagnosis unless otherwise stated.

It is notable that diagnosis is an inherently difficult problem because the solution (search) space grows exponentially with the number of circuit lines, the number of faults and the various fault models : $diagnosis\ space = (\#\ ckt\ lines)^{(\#\ errors)}$. This is because the specification (HDL or the failing chip) is treated as a "black box" controllable at the primary inputs and observable at the primary outputs (Fig. 2). Due to this complexity, development of efficient diagnosis tools remains a challenging task.

Motivated by these observations, we present a *SAT-based solution* to design diagnosis of multiple faults. The formulation is intuitive, straightforward to implement and decouples diagnosis from fault modeling, if necessary. *Model-free* diagnosis is a desirable characteristic for modern devices where fault effects may have a non-deterministic (unmodeled) behavior [2]. However, the method is easily extended to *model-based* diagnosis when fault models are available.

It should be noted that in this work we do not develop a SAT solver, but propose a SAT-based solution to fault diagnosis where existing solvers can be utilized. We argue that SAT naturally captures essential characteristics of diagnosis and structural circuit properties. We also examine different implementation trade-offs and heuristics. To the best of our knowledge, this is the first work to examine design diagnosis using SAT. Experiments with multiple faults demonstrate the efficiency and practicality of the approach.

This paper is organized as follows. Section 2 contains background information and definitions. Section 3 describes the proposed SAT-based formulation and its characteristics. Section 4 contains experiments and the last Section concludes this work.

2 Background

Traditionally, diagnosis techniques are classified as *cause-effect* or *effect-cause* techniques [6]. Cause-effect analysis usually compiles fault dictionaries. Given a failing chip and a set of vectors $v_1, v_2, \ldots, v_k$ from the tester, the chip responses are matched with those in the dictionary to return set of potential faults for each vector. Effect-cause analysis does not use fault dictionaries but simulates input vectors and applies different techniques to identify candidate faults.

In both cases, sets of candidate faults $F_1, F_2, \ldots, F_k$ are returned. When each F_i is injected in the netlist, it explains the (faulty or non-faulty) behavior of test vector v_i alone. These sets are later *intersected* $F = F_1 \cap F_2 \cap \cdots \cap F_k$ to return set F of faults that explains the chip behavior for *all* vectors $v_1, v_2, \ldots, v_k$.

The quality of diagnosis is related to its *resolution*, that is, its ability to return in F the line(s) where fault(s) reside. Due to fault equivalence [6], a solution may not be unique. Ideally, a solution contains only the actual and equivalent fault sites to make it easier for the designer to probe these sites.

In this work, we consider combinational circuits with primitives AND, OR, NOT, NAND, NOR, XOR and XNOR gates and full-scan sequential circuits with a fault-free scan-chain. We use Conjunctive Normal Form (CNF) SAT instances 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 procedure in [7] to translate logic circuits into
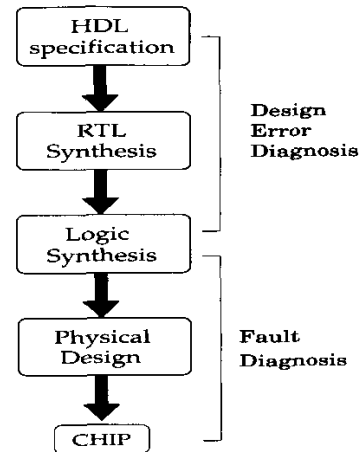

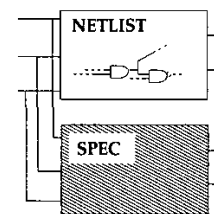Figure 1: Digital VLSI design flow

Figure 2: Fault diagnosis and logic debugging

CNF form. Given a CNF formula, a SAT solver finds a variable assignment that satisfies the formula or it proves that the formula cannot be satisfied.

Without loss of generality, we describe our algorithms on circuits with m primary inputs $X = x_1, x_2, \ldots, x_m$ and a single primary output $y = f(x_1, x_2, \ldots, x_m) = f(X)$. The method is easily generalized to multiple output circuits. We use the names $L = \{l_1, l_2, \ldots, l_n\}$ to represent internal circuit lines including stems and branches. The method in Section 3 adds circuitry to the original circuit. This new hardware requires two extra lines per original circuit line. We use the notation $S = \{s_1, s_2, \ldots, s_n\}$ and $W = \{w_1, w_2, \ldots, w_n\}$ to label these lines.

In this presentation, variables for all circuit lines x_i, l_i, w_i and y are defined to model circuit constraints under simulation for *each* vector v_j. To avoid confusion, we use the notation x_i^j, l_i^j, w_i^j and y^j for these variables and X^j, L^j and W^j for the respective sets (vectors) of variables. Under this notation, superscript j matches the index of simulated test vector v_j. The notation $S = \{s_1, s_2, \ldots, s_n\}$ is used to indicate both variable and line names. Variables for lines S are common to *all* test vectors.

219

3 SAT-based Design Diagnosis

Given a logic netlist and a set of vectors $v_1, v_2, \ldots, v_k$, the algorithm introduces new logic and compiles a CNF formula Φ. This formula has *two components*.

The first component is the conjunction of k CNF formulas $C^j(L^j, W^j, X^j, y^j, S)$, $1 \le j \le k$. Each such C^j enforces *constraints of test vector* v_j on the logic netlist and potential fault sites. Fault locations are encoded in the circuit with extra hardware. The second component $E_N(S)$ encodes *constraints for the cardinality N of injected faults*. These constraints are also coded with new hardware. The value of the number of faults N is a user-specified parameter.

The complete formula Φ is expressed as:

$$\Phi = E_N(S) \cdot \prod_{j=1}^{k} C^j(L^j, W^j, X^j, y^j, S)$$

Intuitively, $\prod_{j=1}^{k} C^j(L^j, W^j, X^j, y^j, S)$ requires that every candidate set of faults satisfies every C^j constraint for all vectors v_j. In other words, faults are *intersected* for all vectors as in traditional diagnosis (Section 2). In the subsections that follow, we describe how to compile each component of Φ with theory and examples for model-free diagnosis. We also discuss memory requirements and propose a number of heuristics to improve run-time performance as well as memory utilization. We finally argue how the proposed methodology can be extended to perform model-based diagnosis.

3.1 Test Vector Constraints

This component of Φ is comprised of k CNF formulas C^j to model circuit and fault location constraints for vector v_j. First, the circuit is modified to reflect the potential presence of faults at various circuit lines. To model the presence of a fault on line l_i, a multiplexer with select line s_i is attached to this line as explained in [13]. This multiplexer is later translated into CNF format and added to the formula.

Consider the circuit in Fig. 3(a), for example. The presence of a fault on line $l = g \to h$ can be represented by a multiplexer with select line s, as shown in Fig. 3(b). The first input of the multiplexer is connected to the output of gate g and the second input of the multiplexer is connected to a new line w to model the potential fault. The output of the multiplexer is connected to the original output of g. Observe that the functionality of the original or faulty circuit is selected when the value of the select line is 0 or 1, respectively [13].

The CNF for the multiplexer logic is given in Fig. 3(c). It can be seen that only 4 clauses are required. Hence, the CNF formula for the complete circuit in Fig. 3(b) is
$C = (x_1 + l') \cdot (x_2 + l') \cdot (x_1' + x_2' + l) \cdot (s + l' + z) \cdot (s + l + z') \cdot (s' + w' + z) \cdot (s' + w + z') \cdot (x_3 + y) \cdot (z + y) \cdot (x_3' + z' + y')$.

Once multiplexers are introduced at every line, the new circuit is translated to CNF. To get the final C^j, we need to insert clauses to represent input/output circuit behavior constraints for the test vector v_j. This can be done with a set of unit-literal clauses for the set of primary input variables $x_1, x_2, \ldots, x_m$ and primary output y. These literals agree with the respective logic values of the vector v_j; that is, if v_j assigns a logic 1 (0) to input x_j then x_i^j ($x_i^{j'}$) appears in the formula and so on.

Example: Recall the circuit in Fig. 3(a) and assume there is a single stuck-at 1 fault on line l. The input test vector $v = (x_1, x_2, x_3) = (1, 0, 1)$ detects the fault as a logic 1 appears at the output of the good circuit while a logic 0 appears at the output of the faulty one. The construction requires unit-literal clauses x_1, x_2, x_3 and y' to be added to C. Hence, the final CNF formula for vector v is $C^v = C \cdot x_1 \cdot x_2' \cdot x_3 \cdot y'$.

This process is repeated for every test vector v_j, $j = 1 \ldots k$ to get CNFs $C^j(L^j, W^j, X^j, y^j, S)$. Note that each such formula requires a new set of variables (and literals) for primary inputs (X^j), primary outputs (y^j), internal circuit lines (L^j) and fault sites (W^j). This is because every input test vector may translate into a different set of constraints for circuit lines and fault locations. However, only one set of select line variables $S = s_1, s_2, \ldots, s_n$ is used because the fault(s) of a solution must satisfy *all* vector constraints simultaneously. The second component, described next, constrains the cardinality N of these faults.
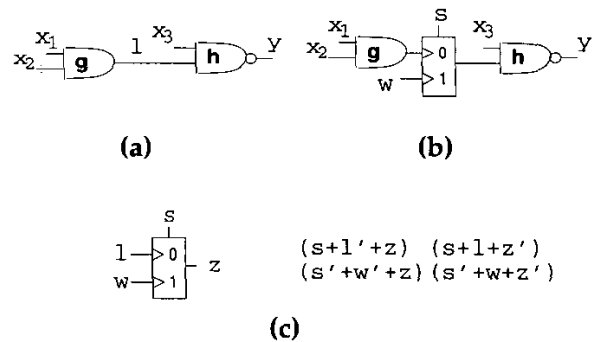

(a) **(b)**

$(s+1'+z)$ $(s+1+z')$
$(s'+w'+z)$ $(s'+w+z')$

(c)

Figure 3: Modeling Fault Sites

3.2 Fault Cardinality Constraints

This component attaches additional hardware to the circuit that enforces constraints $E_N(S)$ and requires a solution set with at most N faults. When the circuit and the added hardware from subsections 3.1 and 3.2 are translated into CNF, we obtain formula Φ. We use an example to provide the intuition for the single fault ($E_1(S)$) case first. Later, we give the hardware construction that generalizes results to multiple faults.

220

Example: Consider the formula C^v computed by the first component. This formula models the circuit in Fig. 3(b) under simulation of test vector $v = (1, 0, 1)$. Assume s (multiplexer select line) is introduced as an additional unit-literal clause so that the formula becomes $C^v = C \cdot x_1 \cdot x_2' \cdot x_3 \cdot y' \cdot s$. Given this new C^v, a SAT solver will attempt to find a satisfying variable assignment for the circuit lines *and* the variable w so that the circuit emulates the faulty chip behavior for vector v. The multiplexer will be forced to select line w and the solver will return $w = 1$ to indicate a stuck-at 1 fault on line l.

The general idea to code $E_N(S)$ is an extension of the example above. That is, formula Φ can be updated with clauses that enumerate exhaustively all possible sets of fault sites. These clauses will enforce subsets $s_{i_1}, s_{i_2}, \ldots, s_{i_N}$ of S to be set to a logic 1 and indicate that N faults are excited. Although this formulation for $E_N(S)$ is intuitive, it requires an exponential number of clauses to be inserted *explicitly* into the formula.

To overcome a memory explosion with increasing values of N, we follow a different approach and encode $E_N(S)$ using the hardware construction shown in Fig. 4. This hardware acts as a counter, forcing the SAT solver to "enumerate" sets of N fault sites. In that figure, thick lines indicate buses of $O(logn)$ bit-width ($N \leq n$) and all other lines represent single bit buses.

The hardware in Fig. 4 performs a bitwise addition of the multiplexer select lines $S = s_1, s_2, \ldots, s_n$ and compares the result to the user-defined number of faults N. The output of the comparator is "forced" to logic 1 with a unit-literal clause so that the bitwise addition of the members of S (that is, the set of fault sites enumerated) is always equal to N. As with the select lines themselves, the variables introduced for this hardware are common to all vectors v_j.

It can be shown that the number of CNF clauses introduced with this hardware construction is linear $O(n)$ but we omit the proof due to lack of space. Intuitively, this *implicit* hardware representation for $E_N(S)$ provides a trade-off between time and space. In the section that follows, we argue that modern SAT-solvers take advantage of this trade-off in practice and that they avoid an exponential explosion in the time domain. Experiments in Section 4 confirm this observation.

3.3 Implementation Details

As explained, a multiplexer requires 4 additional clauses and the counter construction in Fig. 4 is done with $O(n)$ clauses. Therefore, *space requirements* for Φ are *linear* $O(nk)$ in both the number of circuit lines n and the number of vectors k.

Although space efficient, for large industrial circuits the formula Φ may grow quickly with the number of vectors. To further reduce space requirements yet pre-
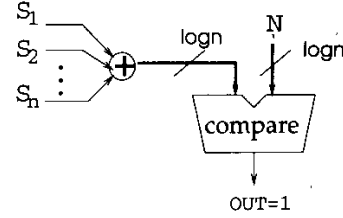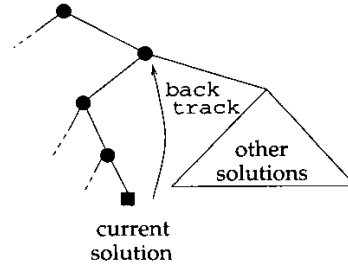

Figure 4: Hardware for multiple errors

Figure 5: Implementation heuristics

serve efficiency, one may compile a set of formulas $\Phi_1, \Phi_2 \ldots, \Phi_{\lceil \frac{k}{p} \rceil}$. Each formula encodes constraints for p distinct test vectors to keep the memory requirements low. Intuitively, Φ is the conjunction of all these formulas Φ_i. To create Φ_i we place multiplexers only on the fault sites that qualified Φ_{i-1}. The rationale of the heuristic lies in the fact that in diagnosis a small number of vectors (such as $p = 10$ vectors) usually screens the majority of invalid candidates [5] [13]. Consequently, only a few fault sites and respective multiplexers (in our experiments less than 10% of the circuit lines on the average) are introduced in subsequent phases of the algorithm. As the discussion in Section 2 indicates, the number of fault sites progressively reduces as i increases.

In the remainder of this subsection, we discuss *time requirements* and *speed-up heuristics*. We also explain why the proposed SAT-based formulation performs (but it is not limited to) model-free diagnosis.

Modern SAT-solvers [9] [11] are enriched with clause-learning and backtracking techniques to help prune the solution space. To take advantage of these techniques, the SAT solver is modified as follows.

For every multiplexer with select line s_i and inputs l_i and w_i, clause $(s_i + w_i^{j'})$ is added for vector v_j to denote the logic implication $s_i' \to w_i^{j'}$. This has the desirable effect that when fault on line l_i is not selected ($s_i = 0$), then the value on w_i is immediately set to logic 0 to prevent unnecessary branching of the SAT solver on w_i.

Additionally, as soon as the solution of fault sites $s_{i_1}, s_{i_2}, \ldots, s_{i_N}$ is returned, the SAT-solver does not reset and start to search for another solution from scratch. Instead, the clause $(s_{i_1}' + s_{i_2}' + \cdots + s_{i_N}')$ is immediately added as a *learned clause*. This is illustrated in

221

Fig. 5 where dotted lines indicate explored portion of the solution space. Upon discovery of a solution, the tool backtracks and may reuse part of the past computation to identify other solution(s) or return unsatisfiability (no additional solutions). This is useful in fault diagnosis where all actual and equivalent solutions need be probed by the test engineer. Experiments show that this heuristic can improve performance substantially.

To improve performance further, the algorithm originally inserts multiplexers only at *structural dominators* [6] of the circuit. This is sufficient for model-free diagnosis since any fault effect on a line covered by a dominator will also be present at the dominator. This has the benefit that the size of the set W is smaller and easier to tackle by the solver. Once a set of dominator-solutions is identified, a second pass is run to find solutions in their respective fan-in cones.

We now elaborate on the model-free/model-based diagnosis nature of the approach. The SAT-based formulation does not make any assumption on the logic value of the fault for each vector v_j. Therefore, it performs model-free diagnosis. This is a desirable characteristic because it may capture faults with "non-deterministic" behavior [2]. However, it is interesting to reason about the logic assignments to variables $w_{i_1}^j, w_{i_2}^j, \ldots, w_{i_N}^j$ on circuit lines $l_{i_1}, l_{i_2}, \ldots, l_{i_N}$ of a solution for all vectors v_j.

As explained, these logic line assignments are required to guarantee that the netlist emulates the behavior of the specification for v_j. The test engineer may use these values to determine the behavior of the fault during fault modeling [6]. Notice that proper reasoning on the values of W^j allows for modeled diagnosis as well. This is achieved if the solver enumerates fault models on the suspicious lines "on the fly" during execution. For example, a stuck-at 1 fault on line l_i is emulated if we set $w_i^j = 1$ for all values (vectors) of j. This can be done by adding $w_i^j, \forall j$, as unit-literal clauses into Φ.

Because of all these desirable characteristics, we conclude that SAT provides an attractive platform for fault diagnosis. Experiments presented in the Section that follow confirm its robustness in practice.

4 Experiments

In this Section we present experiments for a prototype tool implemented with the SAT-solver described in [9]. Experiments are conducted for single and double stuck-at faults in the ISCAS'85 and ISCAS'89 benchmark circuits. We use optimized versions of the ISCAS'85 circuits in order to simulate a typical fault-diagnosis environment. For the ISCAS'89 circuits, we used full-scan versions of the original benchmark circuits. Using the original, non-optimized versions makes the diagnosis process *harder* because of redundancies present in the circuits.

Table 1: Single stuck-at faults

ckt name	# of vectors	# of clauses	# fault sites		CPU (sec)	
			dom.	all	dom.	all
C432	10	11,112	2	5	0.08	0.01
C499	10	28,644	2	8	0.54	0.04
C880	10	22,814	1	6	0.23	0.03
C1355	10	27,904	1	9	0.62	0.04
C1908	10	21,849	2	9	0.23	0.02
C2670	10	36,284	1	30	0.26	0.08
C3540	10	50,731	1	6	1.55	0.07
C5315	10	86,121	2	13	2.27	0.17
C6288	10	173,860	1	5	39.39	0.23
C7552	10	117,928	4	14	3.37	0.19
S13207	20	641,425	2	28	34.88	1.26
S15850	20	728,890	2	16	10.70	1.55
S35932	20	1,500,907	1	12	29.71	2.74
S38417	20	1,758,242	3	17	28.87	7.85
S38584	20	1,765,807	2	10	21.71	7.78

Due to lack of space, we present results only for stuck-at faults. Since most common design errors are modeled in terms of stuck-at faults [1], experiments presented here are expected to be representative for logic debugging as well. The locations of the faults are selected at random. We run experiments on a SUN Blade 100 workstation with 512MB of memory. Ten experiments are performed for each circuit and for each fault case. Average values are reported in the next paragraphs and run-times are in seconds.

Table 1 contains results on single stuck-at faults and Table 2 shows information in a similar manner for double faults. The first column of each table has the circuit name and the second column contains the number k of test vectors used in diagnosis. This set contains mainly vectors with failing responses. Test vector generation is not the subject of this work [7]. The third column has the number of initial clauses for the dominator pass before learned clauses are added. These numbers confirm that memory requirements are linear to circuit size and to the number of vectors. For example, $C432$ requires approximately half the number of clauses of $C880$ because it has nearly half the number of lines.

The number of fault sites returned are found in columns 4 and 5. Column 4 shows the number of fault sites at structural dominators and column 5 shows the number of equivalent fault sites returned. These numbers confirm the accuracy and resolution of the approach. The last two columns have CPU times. Column 6 contains the average runtime per fault location for the dominator step. Column 7 contains the average runtime per fault for each solution in the second pass. The total runtimes for the first and second pass can be determined by multiplying the numbers in columns 4 and 6 and columns 5 and 7 respectively. These numbers suggest that SAT is very efficient at performing diagnosis.

Table 2: Double stuck-at faults

ckt · name	# of vectors	# of clauses	# fault sites		CPU (sec)	
			dom.	all	dom.	all
C432	20	20, 592	4	14	0.25	0.06
C499	20	53, 814	3	19	2.06	0.30
C880	20	42, 644	3	13	1.05	0.17
C1355	20	52, 504	3	13	1.17	0.26
C1908	20	40, 939	10	34	0.51	0.29
C2670	20	69, 004	4	27	0.41	0.31
C3540	20	95, 511	3	11	8.93	0.23
C5315	20	166, 211	2	10	1.70	0.54
C6288	20	324, 560	6	17	76.34	19.72
C7552	20	222, 208	3	16	9.48	1.46
S13207	20	641, 425	3	31	11.01	7.62
S15850	20	728, 890	3	22	34.94	7.40
S35932	20	1, 500, 907	2	31	40.78	19.63
S38417	20	1, 758, 242	3	17	42.12	10.73
S38584	20	1, 765, 807	3	20	44.76	15.48

The benefit of the heuristics in Section 3 is depicted in Fig. 6 that shows the difference in run-times for single faults when they are used. Recall that the first heuristic requires variable w_i^j on line l_i immediately to assume a logic 0 once s_i is not selected for vector v_j. The second heuristic backtracks once a solution is found to reuse previous computation and return the remaining solutions. Run-times indicate that the added clauses allow the SAT solver to prune the solution space.

Experiments demonstrate the effectiveness, flexibility and practicality of the SAT-based solution to design diagnosis. Our formulation can be scaled to larger industrial circuits, by using the approach described in section 3.3. For example, it can be experimentally shown that using two passes of 10 vectors each for S35932 reduces the formula size for each pass to approximately 788,447 clauses with a runtime overhead of only 6% . Moreover, it is possible that in some cases the total runtime may actually be shorter because most of the locations are screened out in the first pass [5] [6] [13].

In the future, we plan to enhance the technique with structural circuit information to improve performance. We also intend to develop diagnosis-specific satisfiability algorithms and techniques that use the information returned by SAT to perform fault modeling.

5 Conclusions

A satisfiability-based formulation of multiple fault diagnosis and logic debugging was presented. The method is intuitive and practical within an industrial environment. Theoretical and experimental results on multiple faults confirm that Boolean satisfiability provides an efficient and effective solution to design diagnosis. This offers new opportunities for satisfiability-based diagnosis tools and diagnosis-specific satisfiability algorithms.
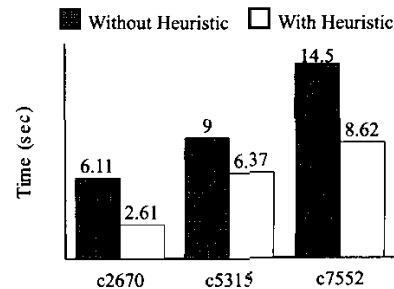

Figure 6: Performance speed up

References

[1] M. S. Abadir, J. Ferguson and T. E. Kirkland, "Logic Verification Via Test Generation," in *IEEE Trans. on CAD*, vol. 7, pp. 138–148, Jan. 1988.

[2] R. C. Aitken, "Modeling the Unmodelable: Algorithmic Fault Diagnosis," in *IEEE Design and Test of Computers, pp. 98-103*, July-Sept. 1997.

[3] P. Bjesse, T. Leonard and A. Mokkedem, "Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers," in *Proc. of CAV, Lecture Notes in Computer Science, Springer-Verlag, vol. 2102, pp. 454-464*, 2001.

[4] E. Goldberg, M. Prasad and R. Brayton, "Using SAT for Combinational Equivalence Checking," in *Proc. of IEEE DATE, pp. 114-121*, 2001.

[5] S. Y.Huang, "Towards the Logic Defect Diagnosis for Partial-Scan Designs," in *Proc. of IEEE ASP-DAC, pp. 313-318*, 2001.

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

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

[8] K. L. McMillan, "Applying SAT Methods in Unbounded Symbolic Model Checking," in *Proc. of CAV, Lecture Notes in Computer Science, Springer-Verlag, vol. 2402, pp. 250-264*, 2002.

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

[10] M. R. Prasad, "Propositional Satisfiability Algorithms in EDA Applications," *Ph.D. Thesis, University of California, Berkeley*, 2001.

[11] 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.

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

[13] A. Veneris and M. S. Abadir, "Design Rewiring Using ATPG," in *Proc. IEEE Trans. on CAD, vol. 21, no. 12, pp. 1469-1479*, Dec. 2002.

[14] R. G. Wood and R. A. Rutenbar, "FPGA Routing and Routability Estimation via Boolean Satisfiability," in *IEEE Trans. on VLSI Systems, vol. 6, no. 2, pp. 222-231*, June 1998.