Journal of Universal Computer Science, vol. 10, no. 12 (2004), 1597-1628
submitted: 26/10/03, accepted: 13/12/04, appeared: 28/12/04 © *J.UCS*

Using Global Structural Relationships of Signals to Accelerate SAT-based Combinational Equivalence Checking[1]

Rajat Arora
(Cadence Design Systems, San Jose, CA, U.S.A.
raarora@cadence.com)

Michael S. Hsiao
(Department of Electrical & Computer Engineering, Virginia Tech, Blacksburg, VA, U.S.A.
hsiao@vt.edu)

Abstract: We propose a novel technique to improve SAT-based Combinational Equivalence Checking (CEC). The idea is to perform a low-cost preprocessing that will statically induce global signal relationships into the original CNF formula of the miter circuit under verification, and hence reduce the complexity of the SAT instance. This efficient and effective preprocessing quickly builds up the implication graph for the miter circuit under verification, yielding a large set of direct, indirect and extended backward implications. These two-node implications spanning the entire circuit are converted into binary clauses, and they are added to the miter CNF formula. The added clauses constrain the search space of the SAT solver and provide correlation among the different variables, which enhances the Boolean Constraint Propagation (BCP). Experimental results on large and difficult ISCAS'85, ISCAS'89 (full scan) and ITC'99 (full scan) CEC instances show that our approach is independent of the state-of-the-art SAT solver used, and that the added clauses help to achieve noteworthy speedup for each of the cases. Also, comparison with Hyper-Resolution (Hypre), Non-Increasing Variable Elimination Resolution (NIVER) and the propositional formula checker HeerHugo, suggests that our technique is more powerful, yielding non-trivial clauses that significantly simplify the SAT instance complexity.

Keywords: Boolean Satisfiability (SAT), Static Logic Implications, Combinational Equivalence Checking (CEC), Propositional Formula, Boolean Formula.
Categories: I.2.6, I.2.8, F.4.m, B.6.2

1 Introduction

In the past four decades, much progress has been made in the field of Boolean Satisfiability (SAT). Due to its numerous Electronic Design Automation (EDA) applications, such as Combinational Equivalence Checking (CEC) [Goldberg 01, Lu 03a, Novikov 03, Silva 99b, Silva 99c], Bounded Model Checking (BMC) [Biere 99, Gupta 03] and Automatic Test Pattern Generation (ATPG) [Larrabee 92, Lu 03b, Stephan 96], SAT continues to be a heavily studied area. The state-of-the-art SAT

[1] *Supported in part by NSF Grants CCR-0196470 and CCR-0305881.*

solvers [Moskewicz 01, Goldberg 02a, Ryan 03] are descendants of the DPLL-algorithm [Davis 62] and usually operate on Boolean formulas represented in Conjunctive Normal Form (CNF). This form consists of the logical AND (conjunction) of clauses, such that each clause is a logical OR (disjunction) of one or more literals. A literal is a variable in its true or complemented form. For the CNF formula to be satisfied, each of the individual clauses should be satisfied (sat). Each clause is also called an implicate of the CNF formula. While trying to satisfy a given CNF formula, a SAT solver makes decisions based on a given set of variable selection heuristics [Goldberg 02a, Moskewicz 01, Silva 99a, ZhangH 97]. It learns dynamically from the conflicts encountered during the search and generates conflict-induced clauses [Goldberg 02a, Moskewicz 01, Silva 99a, ZhangH 97] that can subsequently constrain the search. However, the conflict clauses learnt dynamically have the following disadvantages:

- Not all learned clauses are useful, especially the long clauses.
- The set of all learned clauses can grow very large.
- The clauses are learned gradually over the entire SAT search, which may take a long time.

1.1 Previous Work

Recently, efforts have been made to improve the SAT-based CEC by inducing useful information into the original CNF formula before the SAT solver starts. These efforts have enabled to overcome the above disadvantages to some extent. In [Lu 03a], probable correlations among signal pairs were first obtained by random simulation of the miter circuit. Then, explicit learning was performed wherein the correlated signal pairs were assigned values that would most likely result in a conflict. A SAT solver was then invoked to quickly learn a fixed number of conflict-induced clauses, corresponding to every pair of possibly correlated signals. Because random simulation was used, only a subset of the signal correlations could be identified. In [Novikov 03], the author introduced a technique that involved branching on small subsets of CNF variables, and analyzing the results of unit propagation. A restricted version of this technique was implemented, which focused on deducing constant values and equivalence relationships. In [Li 00], equivalence reasoning was integrated into the Davis-Putnam procedure [Davis 62] to enhance its performance on problems containing equivalence clauses. In [Gupta 03], which focuses on improving SAT-based BMC, local BDDs were used to capture relationships among the Boolean variables of the CNF formula in the form of a characteristic function. The nodes/variables for which BDDs were created were called seed nodes, and these were selected statically or dynamically during the decision phase. Every path leading to the terminal node 0 in the resulting local BDD denoted a conflict, and the negation of the corresponding literals was added as a multi-literal learned clause to the existing CNF formula. However, the locally built BDDs were not helpful in extracting the global relations. In [Kuhelmann 01], the authors integrated BDD-based Boolean reasoning, local structural transformations and circuit-based SAT procedure in one framework, and a shared AND/INVERTER graph representation was used for solving the problem. Probing-based preprocessing techniques for manipulating propositional satisfiability formulae were proposed in [Lynce 03]; meaningful information was inferred from a table of triggering assignments, built by assigning a value to each of

the variables and carrying out unit propagation. The technique also subsumed the additional binary clauses obtained in [Gelder 93]. In [Goldberg 03], the authors integrated the notion of *levels of variables* inside the SAT solver, such that the variables closer to the primary inputs were preferred during decision making. This heuristic considerably improved the SAT solver performance on some of the CEC instances.

In [Groote 00], the authors introduced a propositional formula checker HeerHugo which is based on similar principles as the patented Stalmarck's method [Stalmarck, Sheeran 00]. HeerHugo first converts its input *n*-CNF formula (i.e., a CNF formula with maximum clause length of *n* literals, where *n* is any natural number) into a 3-CNF formula (i.e., a CNF formula where the clauses contain at most 3 literals), and then applies a set of simple rules to determine if it could find some contradiction in the CNF formula. These simple rules are the following:

- unit resolution—this rule identifies any clause with only one literal and assigns a logic 1 to that literal. This assigned value is propagated throughout the CNF formula.
- implication cycle removal—this rule identifies any implication cycle and replaces the literals in an implication cycle by a representative literal. For example, if $p \Rightarrow q$, $q \Rightarrow r$ and $r \Rightarrow p$, then the literals p, q and r are equivalent and form an implication cycle. Therefore, any of the three literals can be used as a representative literal for denoting all of them.
- subsumption checking [Groote 00]—this rule adds a new clause to the CNF formula only after checking if no similar clause already exists in the formula. For example, if a clause $(p \vee q \vee r)$ or $(p \vee q)$ or $(p \vee r)$ or $(q \vee r)$ already exists, then a new inferred clause $(p \vee q \vee r)$ will not be added.
- the classical Davis-Putnam rule [Davis 60]—this rule eliminates propositional variables using binary resolution. In order to eliminate a propositional variable x, the classical Davis-Putnam rule, also called Variable Elimination Resolution (VER) method, forms two sets of clauses Px and Nx with the variable x appearing in positive polarity and negative polarity, respectively. It then performs binary resolution on these two sets of clauses resulting in a set of resolvents R. Finally, it eliminates the variable x by removing all the clauses in $(Px \cup Nx)$ and adding all the clauses in R to the CNF formula.

The Davis-Putnam rule used as a part of the simple rules in HeerHugo was applied in a restricted manner such that a propositional variable was eliminated only when it reduced the CNF formula size or when there was very limited growth. If no contradiction was derived after applying the simple rules, HeerHugo adopted a *branch/merge rule* to prove the satisfiability/ unsatisfiability of the CNF formula. For a CNF formula Φ_{old}, with x as one of the propositional variables, applying the branch merge rule results in a new CNF formula Φ_{new} given by,

$$\Phi_{new} = \Phi_{old} \wedge (C_x \cap C_{\neg x}),$$

where $C_x \equiv \Phi_{old} \wedge x$ and $C_{\neg x} \equiv \Phi_{old} \wedge \neg x$, represents the set of conclusions obtained by applying simple rules with x added to Φ_{old} and $\neg x$ added to Φ_{old}, respectively. This branch/merge rule (called level 0 branch/merge rule) is iteratively applied to all the propositional variables until the intersection between C_x and $C_{\neg x}$ is empty for each of them. If still no contradiction is obtained then the branch/merge rule is applied in a nested way: for example, with the propositional variable x set to *true* in the CNF

formula Φ_{old}, take the intersection of the conclusions obtained by setting the propositional variable *y* first to *true* and then to *false*, independently, and likewise for all the variables (called level 1 branch/merge rule). This way the branch merge rule can be applied at higher levels, which in turn increases the computational complexity of the algorithm. The reader is referred to [Groote 00] for all details.

More recently, in [Bacchus 02, Bacchus 03], preprocessing based on Hyper-Resolution and Equality Reduction was explored. The Hyper-Resolution technique takes as input the following:

(*a*) a single *n*-ary clause ($n \geq 2$), i.e. ($l_1 \vee l_2 \vee l_3 \ldots \vee l_n$), and

(*b*) $n - 1$ binary clauses each of the form ($\neg l_i \vee l$) where ($i = 1, \ldots, n - 1$)

It then produces as output a new *binary clause* ($l \vee l_n$). For example, using Hyper-Resolution on the inputs ($a \vee b \vee c \vee d$), ($h \vee \neg a$), ($h \vee \neg c$), and ($h \vee \neg d$), the new binary clause ($h \vee b$) is produced. Hyper-Resolution is equivalent to a sequence of ordinary resolution steps (i.e., resolution steps involving only two clauses). However, a sequence of ordinary resolution steps would generate clauses of intermediate length while Hyper-Resolution side-steps this to only generate a final binary clause. In a SAT solver it is generally counter-productive to add these intermediate clauses to the CNF database, but it can be very useful to add the final binary clause. The above resolution steps also help to generate *unit clauses* (clauses with only one literal) which further simplify the CNF formula. Their preprocessing algorithm also performs equality reduction if the CNF database has equivalent literals. For example, if the CNF formula contains ($\neg a \vee b$) as well as ($a \vee \neg b$) (i.e, $a \Rightarrow b$ as well as $b \Rightarrow a$), then by equality reduction we can replace *b* with *a*. The steps involved in equality reduction are:

- replace all instances of *b* in the CNF formula by *a*,
- remove all clauses which now contain both *a* and $\neg a$,
- remove all duplicate instances of *a* (or $\neg a$) from all clauses.

This process might generate new unit clauses. The Hyper-Resolution technique was shown to be highly effective on a large variety of SAT benchmarks.

Lately, a SAT preprocessor based on VER method [Davis 60] namely NIVER was introduced in [Subbarayan 04]. This preprocessor is a special case of VER [Davis 60] such that it does not allow an increase in the formula size, with respect to the total number of literals in the original CNF formula Φ. For a propositional variable *x* to be eliminated, NIVER forms two sets of clauses *Px* and *Nx* and finally the set of resolvents *R* (similar to the VER method described before). Now if the total number of literals in ($Px \cup Nx$) is greater than or equal to the total number of literals in *R*, NIVER eliminates the variable *x* by removing all the clauses in ($Px \cup Nx$) from Φ and adding all the clauses in *R* to Φ. Except checking for tautology in the resolvents, NIVER does not do any complex steps like subsumption checking. Unlike Hypre, NIVER does not perform any unit propagation nor does it check for any unit clauses. It continues to iterate until no more variables can be removed from the CNF formula.

1.2 Our Approach

In our approach, unlike [Lu 03a, Novikov 03, Gupta 03] we statically and efficiently identify useful non-trivial relations among signals (variables) over the *entire* miter circuit. We then augment the existing CNF formula by adding these relations as

clauses, before the SAT solver starts. Instead of working on the CNF formula as in [Bacchus 02, Bacchus 03, Lynce 03, Gelder 93, Subbarayan 04], we work on the circuit netlist for inferring additional clauses. The preprocessing step quickly builds the implication graph [Zhao 01] for the miter-circuit under verification. The resulting indirect and extended backward implications help us to deduce pure literals (unit clauses), equivalent literals and other non-trivial implication relations among the CNF variables. The non-trivial implication relationships are converted into two-literal clauses, which are added to the CNF database. These added clauses prune the search space and provide correlation among different variables, which enhances the Boolean Constraint Propagation [Zabih 88, Moskewicz 01, Silva 99a]. Unlike NIVER [Subbarayan 04], we don't do any variable elimination through resolution, nor do we remove any existing clauses from the original CNF database. Instead, we focus on adding many binary clauses which embed in them powerful relations among the CNF variables that are difficult to deduce otherwise. Two state-of-the-art SAT solvers are used in our experiments: *BerkMin* [Goldberg 02a] and *Siege* [Ryan 03]. Experimental results for combinational circuit equivalence checking show that our proposed method is independent of the underlying SAT solver, and we achieve significant speedup in each of the cases. Comparison with the recently developed preprocessing technique hyper-binary resolution [Bacchus 02, Bacchus 03], suggests that our proposed technique is much more powerful and the resulting non-trivial clauses are difficult to obtain using the hyper resolution approach. These new clauses when added to the original CNF formula reduce the SAT instance complexity significantly. The superiority of our technique is further underlined by comparison with the other recent preprocessor NIVER [Subbarayan 04]. We show through experimental results that we outperform NIVER by a huge margin. Also, we compare our results with a propositional formula checker HeerHugo [Groote 00] to further show the effectiveness of our approach.

The rest of the paper is organized as follows. Section 2 gives the background of static implications that we have used in our implementation. Section 3 presents observations when the static implications consisting of indirect and extended backward implications are utilized in a SAT framework. Section 4 discusses the formalization of static implications in the CNF formula. We present a suite of Lemmas and Theorems to prove that the clauses added using static implications are implicates of the CNF formula and preserve its satisfiability. Section 5 gives the implementation algorithm. Experimental results are discussed in Section 6, and Section 7 concludes the paper.

2 Preliminaries

2.1 Static Implications

Static implications are obtained by setting each gate in the Boolean circuit to logic value 1 and 0 independently, and analyzing the result of propagating these values throughout the circuit. An efficient way for representing the implication relations is by using an implication graph where the nodes represent gate with values and edges represent implication relationships [Zhao 01]. For a given circuit with K gates, the

total number of nodes in this graph is $2K$, since each gate can take on a logic value of 0 or 1.

The following terminology is used:

- (N, v): Assign logic value v to gate N where $v \in \{0, 1\}$.
- $(N, v) \rightarrow (M, w)$: Assigning logic value v to gate N implies gate M would be assigned a value w.
- *impl*$[N, v]$: Set of all implications resulting from assigning logic value v to gate N.
- *contrapositive law* [Schulz 88]: If $(N, v) \rightarrow (M, w)$, then the contrapositive law states that $(M, w') \rightarrow (N, v')$, where w' and v' are the complementary values of w and v, respectively. This property can be used to identify additional (possibly non-trivial) implications.
- *impossible/constant nodes*: If $(M, w) \rightarrow (N, v)$ and $(M, w) \rightarrow (N, v')$ or if $(M, w) \rightarrow (M, w')$, then (M, w) is impossible, i.e., gate M would never be able to acquire value w and would be a constant with value w' (for clear understanding refer to Figure 2 and the text under direct implications).
- *transitive law*: If $(M, w) \rightarrow (N, v)$ and $(N, v) \rightarrow (P, u)$, then the transitive law states that $(M, w) \rightarrow (P, u)$.
- *fanins*: fanins of a gate N is the set of adjacent gates driving gate N.
- *fanouts*: fanouts of a gate N is the set of adjacent gates driven by gate N.
- *target gate*: The gate whose implications are being computed by assigning it value v.
- *unjustified gate*: A gate G that has a specified output signal or at least one specified input signal; if the output signal is specified, it is not determined by its inputs/fanins. And if any of the inputs/fanins are specified, they do not determine the gate's output value.
- *unjustified output specified gates*: Subset of unjustified gates whose output value is specified, but is not determined by its inputs/fanins.
- *controlling value*: A logic value at any of the fanins which can determine the gate's output value (see Table 1 for controlling values of different gate types).
- *inversion value*: If the output of the gate is inverted as in the case of NOT, NAND, and NOR gates, the inversion value is 1; otherwise 0 (see Table 1 for inversion values of different gate types).
- *unit-clause rule*: If a clause has n literals and $n - 1$ of its literals have been assigned to logic value 0 by the current state of decision assignments, then the unassigned literal should take on logic value 1 for the CNF formula to be satisfiable. This literal is called a *pure literal* or *implied value*.
- *Boolean Constraint Propagation* (*BCP*) [Zabih 88, Moskewicz 01, Silva 99a]: Applying the unit-clause rule repeatedly until no more pure literals can be obtained.
- *BCP* (x, v): Set of values implied by performing BCP with x assigned to logic value v.

The static logic implications are made up of direct, indirect and extended backward implications. Direct implications can be easily determined whereas indirect and extended backward implications [Zhao 97, Zhao 01] are non-trivial, and their discoveries require combination of simulation, transitive law and contrapositive law [Schulz 88]. The mathematical definitions of direct, indirect and extended backward

implications are given below and the concepts are illustrated using the example circuit shown in Figure 1.

2.1.1 Direct Implications

Direct implications of a gate *G* consist of implications associated with the gates *driving* and *driven* by *G*. Such implications are easily computed by traversing through the immediate fanins and fanouts of the gate. The direct implications are of two types: 1) direct forward implications, and 2) direct backward implications. To compute direct forward implications, a controlling value of *c* at any of the fanins implies a value of *c XOR i* at the gate output, where *i* is the inversion value of the gate. Table 1 gives the controlling value (*c*), the non-controlling value (*nc*) and the inversion value (*i*) for different gates. Note that the non-controlling (*nc*) value is just the complement of the controlling value (*c*). Similarly, to compute direct backward implications, a value of *nc XOR i* at the output implies a value of *nc* at all the fanins.

Gate	Controlling value (*c*)	Non-Controlling value (*nc*)	Inversion value (*i*)
AND	0	1	0
NAND	0	1	1
OR	1	0	0
NOR	1	0	1

Table 1: Controlling, non-controlling and inversion values for various gates

Consider the example circuit in Figure 1. Here, *e* represents an OR gate, *f*, *h* and *k* are NAND gates, *i* and *m* are AND gates, *g* is a NOT gate and *j* is an XOR gate. Now consider gate *f*. When we assert a logic value 0 on its output, the *direct forward implications* are (*g*, 1) and (*h*, 1). The *direct backward implications* are (*e*, 1) and (*c*, 1). Therefore, *impl*[*f*, 0] = {(*f*, 0), (*g*, 1), (*h*, 1), (*e*, 1), (*c*, 1)}.

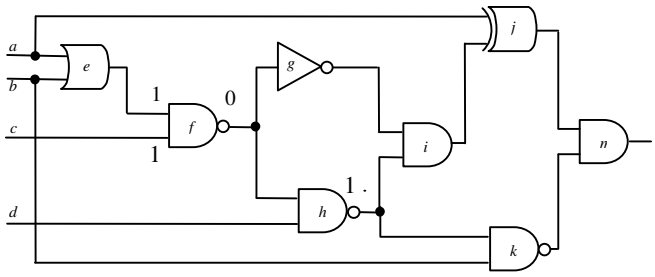

Figure 1: Example circuit

An example circuit showing how direct implications lead to constant nodes is shown in Figure 2. Here, *impl*[*c*, 0] = {(*c*, 0), (*a*, 1), (*b*, 1)}, *impl*[*b*, 1] = {(*b*, 1), (*a*, 0)}, *impl*[*b*, *0*] = {(*b*, 0), (*a*, 1), (*c*, 1)}, *impl*[*a*, 1] = {(*a*, 1), (*b*, 0)} and *impl*[*a*, 0] = {(*a*, 0), (*b*, 1), (*c*, 1)}. Hence, taking transitive closure of (*c*, 0) we get *impl*[*c*, 0] = {(*c*, 0), (*a*, 1), (*b*, 0), (*c*, 1) (*a*, 0), (*b*, 1)}. Since *impl*[*c*, 0] contains both (*a*, 1) and (*a*, 0), therefore (*c*, 0) is impossible and should be constant with value (*c*, 1). We can also

interpret this in a different way. Since, *impl*[*c*, 0] contains (*c*, 1), i.e., (*c*, 0) → (*c*, 1) therefore *c* is a constant with logic value 1.

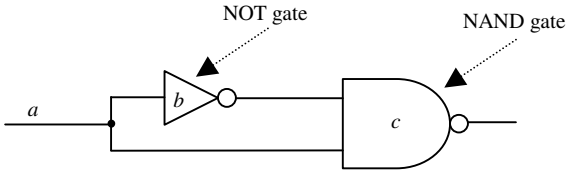

Figure 2: Example circuit illustrating constant/impossible nodes

2.1.2 Indirect Implications

The indirect implications of a node are computed by applying the gate values pertaining to all its direct implications to the circuit netlist and performing logic simulation. All gates, where the output value changes from a don't-care to logic 0 or 1, form the indirect implications of the node. Mathematically, $impl[N, v] = impl[N, v] \cup [LogicSimulate(impl[N, v])]$. Here *LogicSimulate*() refers to performing logic simulation with the implications applied to the circuit. Note that $LogicSimulate(impl[N,v]) \not\subset impl[N, v]$. In fact, the above expression must be interpreted as $impl[N, v]_{new} = impl[N, v]_{old} \cup LogicSimulate(impl[N, v]_{old})$. We follow this convention throughout the manuscript.

Consider the direct implications of (*f*, 0) in Figure 1. We see that (*g*, 1) or (*h*, 1) individually do not imply anything on gate *i*. However, together they imply (*i*, 1). Therefore, (*f*, 0) → (*i*, 1) is an indirect implication and can be computed by a simple logic simulation of the list *impl*[*f*, 0]. These indirect implications are added to the implication graph of the circuit along with their corresponding contrapositive implications. Thus, *impl*[*f*, 0] = {(*f*, 0), (*g*, 1), (*h*, 1), (*e*, 1), (*c*, 1), (*i*, 1)}. These indirect implications have been used in the past with the name of *global implications* and *non-local implications*. Schulz et al. in [Schulz 88] utilized these non-local implications to improve the performance of ATPG engine and later Larrabee [Larrabee 92] and Stephan et al. [Stephan 96], respectively, used them for combinational test generation in a SAT framework.

2.1.3 Extended Backward Implications

The extended backward implications were first introduced by Zhao et al. in [Zhao 97]. These implications are computed by considering (1) the *target gate*, and (2) the *unjustified output specified gates* in the implication list of the target gate.

Let (*G*, *v*) ∈ *impl*[*N*, *v*], and suppose gate *G* has *p* inputs among which *m* inputs (l_1, …, l_m) are unspecified. Here *N* is the target gate and *G* is the unjustified gate with specified *output.*

Case 1: G is an AND gate:

If (*G*, 0) ∈ *impl*[*N*, *v*] and (l_j, 0) ∉ *impl*[*N*, *v*], (*j* = 1, 2, …, *m*), then

$impl[N, v] = impl[N, v] \cup [\bigcap_{i=1}^{m} LogicSimulate(impl[N, v] \cup impl[l_i, 0])]$

The above mathematical formulation states that if the implication set of (N, v) contains an AND gate G which is unjustified output specified (i.e., it has an output value of 0 which is not determined by the value of its fanins), then the common set of implications obtained by setting each of the unspecified fanins to 0 under the current assignment of (N, v), will be appended to the implication set of (N, v).

Case 2: G is an OR gate:

If $(G, 1) \in impl[N, v]$ and $(l_j, 1) \notin impl[N, v]$, $(j = 1, 2, \ldots, m)$, then

$impl[N, v] = impl[N, v] \cup [\bigcap_{i=1}^{m} LogicSimulate(impl[N, v] \cup impl[l_i, 1])]$

The above mathematical formulation states that if the implication set of (N, v) contains an OR gate G which is unjustified output specified (i.e., it has an output value of 1 which is not determined by the value of its fanins), then the common set of implications obtained by setting each of the unspecified fanins to 1 under the current assignment of (N, v), will be appended to the implication set of (N, v).

In the same way, extended backward implications can be computed for *NAND* and *NOR gates.*

Case 3: G is a 2-input XOR gate:

If $(G, 1) \in impl[N, v]$ and both inputs l_0 and l_1 are unspecified, then,

$impl[N, v] = impl[N, v] \cup \{LogicSimulate(impl[N, v] \cup impl[l_0, 0] \cup impl[l_1, 1]) \cap LogicSimulate(impl[N, v] \cup impl[l_0, 1] \cup impl[l_1, 0])\}$

The above mathematical formulation states that if the implication set of (N, v) contains an XOR gate G which is unjustified output specified (i.e., it has an output value of 1 which is not determined by its fanins), then the common set of implications obtained by setting its two fanins to logic value 0 and 1 and then to 1 and 0, respectively, under the current assignment of (N, v), will be appended to the implication set of (N, v).

If $(G, 0) \in impl[N, v]$ and both inputs l_0 and l_1 are unspecified then,

$impl[N, v] = impl[N, v] \cup \{LogicSimulate(impl[N, v] \cup impl[l_0, 0] \cup impl[l_1, 0]) \cap LogicSimulate(impl[N, v] \cup impl[l_0, 1] \cup impl[l_1, 1])\}$

The above mathematical formulation states that if the implication set of (N, v) contains an XOR gate G which is unjustified output specified (i.e., it has an output value of 0 which is not determined by its fanins), then the common set of implications obtained by setting both the fanins to logic value 0 and then to 1, under the current assignment of (N, v), will be appended to the implication set of (N, v).

Case 4: G is a 2-input XNOR gate:

If $(G, 0) \in impl[N, v]$ and both inputs l_0 and l_1 are unspecified then,

$impl[N, v] = impl[N, v] \cup \{LogicSimulate(impl[N, v] \cup impl[l_0, 0] \cup impl[l_1, 1]) \cap LogicSimulate(impl[N, v] \cup impl[l_0, 1] \cup impl[l_1, 0])\}$

If $(G, 1) \in impl[N, v]$ and both inputs l_0 and l_1 are unspecified then,

$impl[N, v] = impl[N, v] \cup \{LogicSimulate(impl[N, v] \cup impl[l_0, 0] \cup impl[l_1, 0]) \cap LogicSimulate(impl[N, v] \cup impl[l_0, 1] \cup impl[l_1, 1])\}$

Since we are dealing with miter circuits, the extended backward implications pertaining to XOR/XNOR gates help to identify many powerful implications, which in turn play an important role in proving the equivalence of the two circuits.

To illustrate the concept of extended backward implications, consider again the example circuit of Figure 1. We see that $impl[f, 0] = \{(f, 0), (g, 1), (h, 1), (e, 1), (c, 1), (i, 1)\}$. The implication list of $(f, 0)$ contains $(e, 1)$ and the OR gate e is unjustified with a specified output. Now justifying $e = 1$ by setting the fanin $a = 1$ yields XOR gate $j = 0$ and $j = 0 \rightarrow m = 0$. On the other hand, justifying $e = 1$ by setting the fanin $b = 1$ results in NAND gate $k = 0$ and $k = 0 \rightarrow m = 0$. Thus, if the OR gate e is justified by any of the fanins, we get a common implication $m = 0$. Therefore, $f = 0 \rightarrow m = 0$ is an extended backward implication of $(f, 0)$, and is appended to the list $impl[f, 0]$. These extended backward implications help to identify the *hard-to-find implications*, and hence are effective for various applications such as capturing additional untestable faults [Zhao 97, Zhao 01].

3 Application of Static Implications to SAT

When a circuit netlist is converted into its equivalent CNF-form, the resulting formula is devoid of global structural information. Also, the topological ordering among the signals is lost. All the internal signals in the original circuit become primary inputs (variables) in the two-level OR-AND CNF formula. As a result, the SAT solver heuristically picks up a variable for decision, without having much information about its impact on future decisions. For example, successive decisions on two different variables might be correlated in some way, but due to absence of global relationships, these variables may be assigned values that may eventually lead to a conflict in the future. In our approach, we try to induce structural relationships into the CNF formula of the miter circuit under verification, such that conflicts are either completely avoided or can be deduced early in the decision process. We first compute the implications on the circuit netlist, and then convert these implications into clause form. These clauses when added to the original CNF formula induce signal correlations among the variables, which in turn accelerates the SAT solver performance.

3.1 Enhanced Boolean Constraint Propagation

Consider again the example circuit of Figure 1. Its CNF formula is shown below. The CNF formula derivation is straightforward and the reader is referred to [Tseitin 68, Larrabee 92] for all details.

$(\neg a \vee e)\ (\neg b \vee e)\ (\neg e \vee a \vee b)\ (f \vee e)\ (f \vee c)$
$(\neg f \vee \neg e \vee \neg c)\ (\neg f \vee \neg g)\ (f \vee g)\ (\neg f \vee \neg d \vee \neg h)\ (f \vee h)$
$(d \vee h)\ (g \vee \neg i)\ (h \vee \neg i)\ (\neg g \vee \neg h \vee i)\ (\neg j \vee a \vee i)$
$(\neg j \vee \neg a \vee \neg i)\ (j \vee \neg a \vee i)\ (j \vee a \vee \neg i)\ (h \vee k)\ (b \vee k)$
$(\neg h \vee \neg b \vee \neg k)\ (j \vee \neg m)\ (k \vee \neg m)\ (m \vee \neg j \vee \neg k)$

In this CNF formula, the clauses $(\neg a \vee e)(\neg b \vee e)(\neg e \vee a \vee b)$ represent the OR gate e, $(f \vee e)(f \vee c)(\neg f \vee \neg e \vee \neg c)$ represent the NAND gate f, $(\neg f \vee \neg g)(f \vee g)$ correspond to NOT gate g, $(\neg j \vee a \vee i)(\neg j \vee \neg a \vee \neg i)(j \vee \neg a \vee i)(j \vee a \vee \neg i)$ correspond to XOR gate j and so on.

Now, let us suppose that the SAT solver heuristically makes the first decision $i = 0$. On assigning $i = 0$ and performing BCP, no unit clauses are obtained. However, from our implication engine, we know that $f = 0 \rightarrow i = 1$, and by contrapositive law $i = 0 \rightarrow f = 1$. The two-literal clause corresponding to this implication is $(i \vee f)$. If we add this clause beforehand to the original CNF formula, setting $i = 0$ will imply $f = 1$ immediately, which in turn will imply $g = 0$. Therefore, learning the information $i = 0 \rightarrow f = 1$, helps us to satisfy a total of 10 clauses instead of satisfying only 4. This is illustrated in Figure 3.

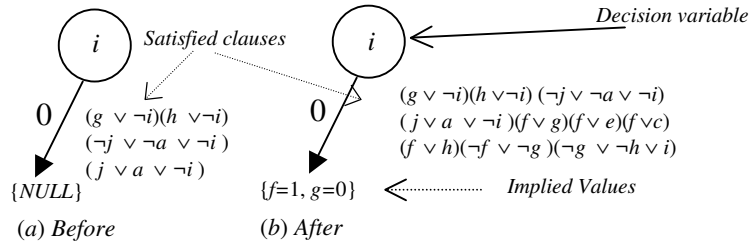

Figure 3: Implied values and satisfied clauses in the CNF formula, before and after adding the clause $(i \vee f)$

3.2 Identification of Equivalent/Complement Literals

The basis of Combinational Equivalence Checking (CEC) is to identify equivalent signals in the two circuits incrementally, proceeding from the primary inputs towards the primary outputs. In SAT-based CEC, identification of such equivalent signals helps to reduce the problem complexity; a decision on one of the signals in the equivalent pair implies a value on the other corresponding signal, which in turn enhances the BCP and reduces the number of decisions required to prove the satisfiabilty/unsatisfiabilty of the CNF formula. The implication graph that we build (as a preprocessing step) for the miter circuit under verification helps us to identify these equivalent signals, which are in turn added as binary clauses to the existing CNF database.

Consider below the CNF formula for the circuit shown in Figure 4:

$$(x \vee p)(y \vee p)(\neg x \vee \neg y \vee \neg p)(x \vee q)(y \vee q)(\neg x \vee \neg y \vee \neg q)$$

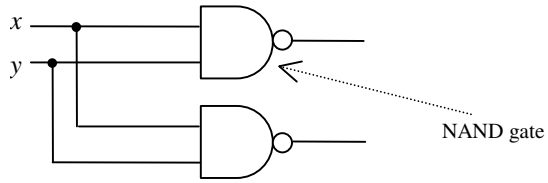

Figure 4: Equivalent/Complement literal identification

We can see that the decision $p = 0$ on unit propagation implies $x = 1$, $y = 1$, and finally $q = 0$. Similarly, the decision $q = 0$ implies $x = 1$, $y = 1$, and finally $p = 0$. But $p = 1$ implies nothing on q; likewise, $q = 1$ implies nothing on p. Hence, we cannot deduce that the two signals p and q are equivalent. However, our implication engine can deduce this relation: $impl[p, 0] = \{(p, 0), (x, 1), (y, 1), (q, 0)\}$, where $(p, 0) \rightarrow (q, 0)$ is an indirect implication. By the contrapostive law, $(q, 1) \rightarrow (p, 1)$. Similarly, $impl[q, 0] = \{(q, 0), (x, 1), (y, 1), (p, 0)\}$, such that $(q, 0) \rightarrow (p, 0)$ is an indirect implication. Again, using the contrapositive law, $(p, 1) \rightarrow (q, 1)$. Thus, $p \leftrightarrow q$. Therefore, for the two indirect implications, $(p, 0) \rightarrow (q, 0)$ and $(q, 0) \rightarrow (p, 0)$, we add up the clauses $(p \vee \neg q)$ and $(q \vee \neg p)$, respectively. The addition of such two clauses proves the equivalence of two variables p and q. It should be noted that every two-literal clause we add embeds in itself both the indirect implication as well as its contrapostive. Similar to equivalent literals, our approach can also identify complementary signals in the circuit. These relations between intermediate points of the circuit are propagated in the forward direction and help to identify additional relations and implications throughout the circuit.

3.3 Identification of Constant/Impossible Nodes

In order to prove the equivalence of two circuits, the corresponding primary outputs of the two circuits are XOR-ed (i.e., a miter circuit is created), and the XOR outputs are checked if they are at constant 0 value. In our approach, building the implication graph for the miter circuit under verification may deduce a few XOR outputs to be constant at logic 0. This happens whenever implications of the following type are obtained:

a. $(Z, 1) \rightarrow (Y, 0)$ and $(Z, 1) \rightarrow (Y, 1)$ or

b. $(Z, 1) \rightarrow (Z, 0)$,

Here Y and Z can be any pair of signals in the miter circuit. The implication of type ***a*** suggests that when Z is set to logic value 1, Y must take on both 0 and 1 as logic values. This is impossible since Y cannot be both 0 and 1 simultaneously. Hence, $Z = 1$ must be impossible, indicating that Z should always be a constant with logic value 0. Similarly, the implication of type ***b*** suggests that $Z = 1$ implies $Z = 0$, i.e., a conflict on itself. This again suggests that $Z = 1$ is impossible and Z has to be a constant with logic value 0. After the implication graph for the miter circuit under verification has been built, all the nodes identified as constants are added as unit clauses (pure literals)

to the original CNF database. This in turn prunes the search space of the SAT solver engine, thereby enhancing its performance.

3.4 Significance of Extended Backward Implications

The concept of extended backward implications helps us to learn some very useful, non-trivial two-node implications. When added as two-literal clauses to the original CNF formula, they play a significant role. We will illustrate this by means of the example circuit in Figure 1. The corresponding CNF formula for this circuit has been given earlier. Now, suppose our objective is to satisfy $m = 1$. Let us assume that the SAT solver makes the following decisions: $m = 1$ (given objective), followed by $f = 0$, and then $a = 0$. (Note that different SAT solvers make decisions based on different heuristics, and hence the set of decisions may vary from one SAT solver to another. We assume these decisions just to explain the efficacy of our technique.) However, we can see that assigning $a = 0$ results in a conflict. Also, on backtracking $a = 1$ yields a conflict. The SAT solver again backtracks and sets $f = 1$, and finally the decisions $d = 0$, $b = 0$ make the formula satisfiable. The resulting decision tree is shown in Figure 5.

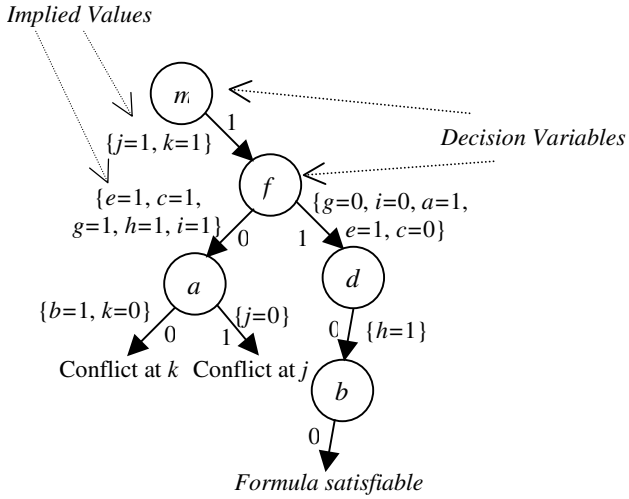

Figure 5: Decision tree without adding any clauses

Now, we use our implication engine as a preprocessing step. From extended backward implications, we learned that $f = 0 \rightarrow m = 0$. Applying the contrapostive law, we obtain $m = 1 \rightarrow f = 1$. Hence, we statically insert the clause $(f \vee \neg m)$ in the original CNF formula. Now, if we ask the SAT solver to satisfy the objective $m = 1$, then $f = 1$ will be implied immediately, and our decision tree will be as shown in Figure 6. We see that adding the two-literal clause results in fewer decisions with no backtracks, and at the same time improves the BCP.

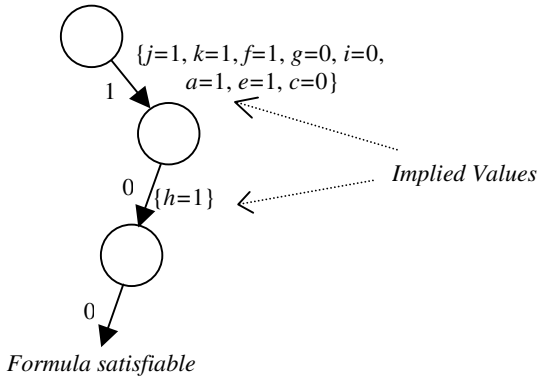

Figure 6: Decision-tree after adding the two-literal clause ($f \vee \neg m$), derived using extended backward implication.

3.5 Comparison of our method with Hypre

We compared our preprocessing technique with the Hyper-Resolution technique introduced in [Bacchus 03]. We observed that their tool Hypre can only deduce a subset of the clauses deduced by our method. This was experimentally verified by running Hypre [Bacchus 03] on the example circuit of Figure 1. It was observed that Hypre was not able to deduce the two-literal clause ($f \vee \neg m$). We then ran Hypre on another example circuit shown in Figure 7. In this case, our preprocessing tool deduced *six* additional non-trivial clauses. On the other hand, Hypre deduced only three clauses. All clauses deduced by our method are listed below, in which only half of them (3 clauses) were obtained by Hypre:

- ($\neg c \vee g$), ($f \vee i$), ($\neg f \vee k$) *were deduced by Hypre as well*
- ($f \vee \neg m$), ($p \vee \neg k$), ($p \vee \neg a$) *were deduced only by our preprocessing tool.*

Here, the clause ($f \vee i$) is obtained by computing indirect implications for node (f, 0), the clauses ($\neg c \vee g$), ($\neg f \vee k$) and ($f \vee \neg m$) are deduced by computing extended backward implications for nodes (g, 0), (k, 0) and (f, 0), respectively. And finally the above implication relations help to deduce the non-trivial clauses ($p \vee \neg k$) and ($p \vee \neg a$), by performing extended backward implications on (p, 0). This corroborates the fact that our technique is more powerful than Hypre, since more implications can be obtained by our method. In Section 6, we give more experimental results, which further underpin the superiority of our technique.

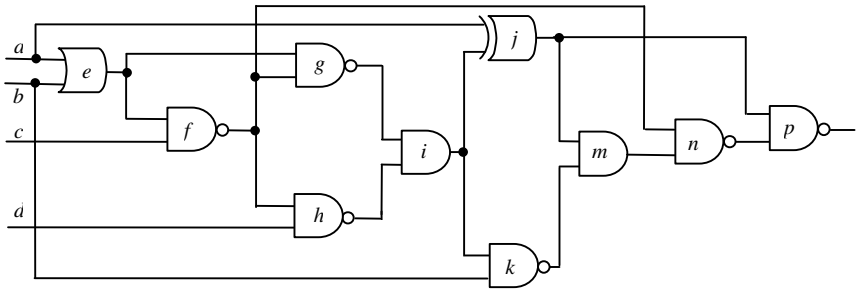

Figure 7: Second example circuit

3.6 Related Work

In [Silva 99b, Silva 99c], the Recursive Learning technique [Kunz 92, Kunz 93] was incorporated into SAT solvers and applied to combinational equivalence checking. The Recursive Learning technique is guaranteed to find all possible necessary assignments in the circuit, given enough levels of recursion. However, as the depth of recursion increases, the time to compute the implications increases exponentially. As a result, in [Silva 99b, Silva 99c], the authors preprocessed the CNF formula using only depth one in Recursive Learning [Kunz 92, Kunz 93]. The depth one Recursive Learning is different from extended backward implications [Zhao 97] used in our approach. Recursive Learning of depth one is equivalent to performing only *direct backward implications* on each of the fanins of the unjustified output specified gates, and determining the common set of implications. On the other hand, extended backward implications make use of the following:

- the implication list of the *target gate* (this implication list includes the *unjustified output specified gates* and their corresponding implications), and
- the implication list of the fanins of the *unjustified output specified gates*

It then performs logic simulation on *both* implication lists to determine the common set of assignments. Extended backward implications help to *quickly* identify the powerful non-trivial implications which may require more than one level of recursion by the Recursive Learning procedure [Kunz 92, Kunz 93].

4 Formalizing Static Implications in the CNF Formula

In this section, we provide a suite of lemmas and theorems that help us to formalize static implications consisting of direct, indirect and extended backward implications in the CNF formula. These lemmas and theorems infer additional clauses that are a superset of the clauses deduced using static implications. Thus, we show that these implications when added as two-literal clauses to the existing CNF database will preserve the satisfiability of the CNF formula. The CNF formula Φ for the example circuit of Figure 1 is shown below.

$\omega_1 = (\neg a \vee e)$	$\omega_2 = (\neg b \vee e)$	$\omega_3 = (\neg e \vee a \vee b)$	$\omega_4 = (f \vee e)$
$\omega_5 = (f \vee c)$	$\omega_6 = (\neg f \vee \neg e \vee \neg c)$	$\omega_7 = (\neg f \vee \neg g)$	$\omega_8 = (f \vee g)$
$\omega_9 = (\neg f \vee \neg d \vee \neg h)$	$\omega_{10} = (f \vee h)$	$\omega_{11} = (d \vee h)$	$\omega_{12} = (g \vee \neg i)$
$\omega_{13} = (h \vee \neg i)$	$\omega_{14} = (\neg g \vee \neg h \vee i)$	$\omega_{15} = (\neg j \vee a \vee i)$	$\omega_{16} = (\neg j \vee \neg a \vee \neg i)$
$\omega_{17} = (j \vee \neg a \vee i)$	$\omega_{18} = (j \vee a \vee \neg i)$	$\omega_{19} = (h \vee k)$	$\omega_{20} = (b \vee k)$
$\omega_{21} = (\neg h \vee \neg b \vee \neg k)$	$\omega_{22} = (j \vee \neg m)$	$\omega_{23} = (k \vee \neg m)$	$\omega_{24} = (m \vee \neg j \vee \neg k)$

Note that each of the gates in the example circuit of Figure 1 corresponds to the propositional variables in the CNF formula shown above.

4.1 Direct implications in the CNF formula

As described in Section 2.1.1, direct implications of a gate x consist of implications associated with the gates *directly* connected to x. For the propositional variable x in the CNF formula Φ, these directly connected gates can be interpreted as all the propositional variables which occur with x in all the clauses. We call these propositional variables *propositional variables directly affected* by x. Unlike the circuit netlist there is no notion of direct *forward* or direct *backward* implications in the CNF formula, since there is no circuit structure. Here, direct implications of a variable x can be interpreted as the implications obtained on propositional variables directly affected by x. The direct implications of the variable x set to logic value 1 or 0 are the values implied after *single* application of the unit-clause rule to the CNF formula Φ after setting x to 1 or 0. This in turn will mean the implications obtained on propositional variables directly affected by x. For example in the CNF formula Φ, for the propositional variable f, the directly affected propositional variables are $\{e, c, g, d, h\}$. Now when f is set to logic value 0, the values implied by application of unit-clause rule to clauses ω_4, ω_5, ω_8, ω_{10} are $(e, 1)$, $(c, 1)$, $(g, 1)$ and $(h, 1)$. These values are in immediate compliance with the direct forward and direct backward implications obtained when the gate f is set to logic value 0 in Section 2.1.1. Note that these implications are already embedded in the original CNF formula, and no new clauses need to be added.

4.2 Formalizing indirect implications in the CNF formula

We firstly give a lemma and a theorem which helps us to infer additional clauses in the CNF formula, and then finally through an observation we show how these inferred clauses form a superset of the clauses obtained using indirect implications.

Lemma 1 [Lynce 03]: Given a CNF formula Φ, if $(y, 1) \in \text{BCP}(x, 1)$, then the clause $(\neg x \vee y)$ is an implicate of Φ.

Proof: The clause $(\neg x \vee y)$ results in two cases:

1. $(x, 1) \rightarrow (y, 1)$: This has been given to us since $(y, 1) \in \text{BCP}(x, 1)$
2. $(y, 0) \rightarrow (x, 0)$: This is obtained by applying contrapositive law to the first case $(x, 1) \rightarrow (y, 1)$

Hence, the clause $(\neg x \vee y)$ can be safely added to the CNF formula Φ under the condition $(y, 1) \in \text{BCP}(x, 1)$. The added clause will always preserve the satisfiability of the CNF formula. □

Theorem 1 [Lynce 03]: Given a CNF formula Φ, if $(y_i, 1) \in \text{BCP}(x, 1)$, $i = 1, 2, \ldots, n$, then each clause of the form $(\neg x \vee y_i)$, $i = 1, 2, \ldots, n$, is an implicate of Φ.
Proof: The theorem directly follows from *Lemma* 1. If a single clause $(\neg x \vee y)$ is an implicate of Φ under the condition $(y, 1) \in \text{BCP}(x, 1)$, then all the clauses $(\neg x \vee y_i)$ where $i = 1, 2, \ldots, n$ are implicates of Φ. □

Observation 1: The set of clauses obtained by Theorem 1 fully subsumes all the clauses obtained using indirect implications.

Indirect implications of a gate G set to value v are obtained by performing logic simulation with direct implications of (G, v) applied to the circuit. This is similar to doing BCP (i.e., repeated application of the unit-clause rule to the CNF formula) when the CNF variable G is set to value v. For example, when f is set to logic value 0 in Φ, $\text{BCP}(f, 0) = \{(f, 0), (e, 1), (c, 1), (g, 1), (h, 1), (i, 1)\}$, where $(e, 1)$, $(c, 1)$, $(g, 1)$, $(h, 1)$, and $(i, 1)$, are obtained from clauses ω_4, ω_5, ω_8, ω_{10} and ω_{14}, respectively. Using Theorem 1, we can add up the clause $(f \vee i)$ to Φ. Now, we see that the above values and hence the clause $(f \vee i)$ are in immediate compliance with the values and the clause obtained using indirect implications when the gate f is set to logic value 0 in Section 2.1.2. However, it must be noted that *LogicSimulate*(*impl*[x, v]) is a proper subset of BCP(x, v), since we can only imply new values in the forward direction using logic simulation. On the other hand, in BCP we don't have any notion of forward/backward directions and the logic reasoning using BCP might lead to some more implications that logic simulation cannot yield. An example is a 2-input AND gate with its output and one of its inputs set to 1. In such a case, BCP will deduce that the other input is also 1, whereas *LogicSimulate*() cannot figure that out. Thus, all the clauses obtained by BCP(G, v) using Theorem 1 will subsume the indirect implications obtained with (G, v). Thus, the above observation will always hold. □

4.3 Formalizing extended backward implications in the CNF formula

We firstly give two lemmas and a theorem which helps us to infer additional clauses in the CNF formula, and then finally through an observation we show how these inferred clauses form a superset of the clauses obtained using extended backward implications.

Lemma 2 [Lynce 03]: Given a CNF formula Φ, for any clause $\omega = (l_1 \vee l_2 \vee \ldots \vee l_n) \in \Phi$, if $(y, 1) \in [\cap^n{}_{i=1}(\text{BCP}(l_k, 1))]$, then $(y, 1)$ will be a necessary assignment of Φ.

Proof: We are given the following:

- Clause ω has n literals, i.e. $\omega = (l_1 \vee l_2 \vee \ldots \vee l_n)$, and
- BCP $(l_1, 1)$ implies $(y, 1)$, (1)
- BCP $(l_2, 1)$ implies $(y, 1)$, (2)
- . . .
- BCP $(l_n, 1)$ implies $(y, 1)$ (n)

We prove this Lemma by contradiction.
Suppose that $(y, 1)$ is not an implicate of Φ. In other words, there exists a satisfying assignment to the CNF formula with $(y, 0)$. However, using equations (1) to (n) by the contrapositive law we obtain $(y, 0) \rightarrow (l_1, 0)$, $(y, 0) \rightarrow (l_2, 0)$, …, $(y, 0) \rightarrow (l_n, 0)$. Since, $(y, 0)$ implies each of the literals $l_1, l_2, \ldots, l_n$ to logic 0, the clause ω would evaluate to 0, causing the CNF formula to become unsatisfiable. Hence, our assumption is wrong and the assignment $(y, 0)$ is not possible. Therefore, $(y, 1)$ is an implicate of Φ. □

Lemma 3: Given a CNF formula Φ, for any clause $\omega = (l_1 \vee l_2 \vee ... \vee l_n) \in \Phi$, if under the assignment $(x, 0)$, the literals $l_1, l_2, \ldots, l_j$ $(j \prec n)$ are implied to 0, and if $(y, 1) \in [\cap^{n}_{k=j+1} \text{BCP}\ (l_k = 1 \text{ and } x=0)\ ]$, then $(x \vee y)$ will be an implicate of Φ.

Proof: We know that for the original CNF formula Φ to be satisfied, every clause $\omega \in \Phi$ needs to be satisfied. If the current assignment $(x, 0)$ causes the literals $l_1, l_2, ..., l_j$ $(j \prec n)$ of ω to evaluate to 0, the clause ω can only be satisfied if any of its remaining literals evaluates to logic 1. Therefore, the lemma states that the common assignment obtained by setting each of the remaining literals to logic 1 will be a necessary assignment under the condition $(x, 0)$. In other words, $(x, 0) \rightarrow (y, 1)$ in Φ, or $(x \vee y)$ is an implicate of Φ.

We continue the proof by contradiction. It is given that the assignment $(x, 0)$ results in the following:

- $l_1, l_2, \ldots, l_j$ are implied to 0, and
- BCP ($l_{j+1} = 1$ and $x = 0$) implies $(y, 1)$ (1)
- BCP ($l_{j+2} = 1$ and $x = 0$) implies $(y, 1)$ (2)
- . . .
- BCP ($l_n = 1$ and $x = 0$) implies $(y, 1)$ $(n - j)$

Applying contrapositive law to equations (1) to $(n - j)$ will yield the following constraints:

- $(y \vee x \vee \neg l_{j+1})$
- $(y \vee x \vee \neg l_{j+2})$
- . . .
- $(y \vee x \vee \neg l_n)$

Suppose, $(x \vee y)$ is not an implicate of Φ. This means that x and y can be 0 simultaneously. Now, when $(x, 0)$ and $(y, 0)$ hold together, the above constraints will cause the literals $l_{j+1}, l_{j+2}, \ldots, l_n$ to be implied to logic 0. Also, the assignment $(x, 0)$ already implies $l_1, l_2, \ldots, l_j$ to logic 0 (given). Thus, the clause ω would evaluate to 0 and the CNF formula will become unsatisfiable. Hence, our contradiction statement is false and x and y cannot be 0 simultaneously. Therefore, $(x \vee y)$ is an implicate of Φ.□

The Lemma 3 is an extension of Lemma 2 and states that if the current assignment $(x, 0)$ implies the literals $l_1, l_2, \ldots, l_j$ $(j \prec n)$ of ω to logic 0, then the common assignment $(y, 1)$ obtained by setting each of the remaining literals of ω to 1, together with the current assignment $(x, 0)$ will result in an implicate $(x \vee y)$ of Φ.

Theorem 2: Given a CNF formula Φ, for any clause $\omega = (l_1 \vee l_2 \vee \ldots \vee l_n) \in \Phi$, if under the assignment $(x, 0)$, the literals $l_1, l_2, \ldots, l_j$ $(j \prec n)$ are implied to 0, then for every $(y_i, 1) \in [\cap^{n}_{k=j+1} \text{BCP}\ (l_k = 1 \text{ and } x = 0)\ ]$, $i = 1, 2, \ldots, m$, $(x \vee y_i)$ is an implicate of Φ.

Proof: The theorem directly follows *Lemma 3*. If a single clause $(x \vee y)$ is an implicate of Φ when $(y, 1) \in [\cap^{n}_{k=j+1} \text{BCP}\ (l_k = 1 \text{ and } x = 0)\ ]$, then all the clauses $(x \vee y_i)$ where $i = 1, 2, \ldots, m$ are implicates of Φ. □

Observation 2: The set of clauses obtained by Theorem 2 is a superset of all the clauses obtained through extended backward implications.

Consider the CNF formula Φ for the example circuit of Figure 1. We assign $(f, 0)$, perform BCP $(f, 0)$ and get the following implications: $\{(e, 1), (c, 1), (g, 1), (h, 1), (i, 1)\}$. Using Theorem 2, we see that along with other clauses, the clause $\omega_3 = (\neg e \vee a \vee b)$ is one of the affected clauses under the assignment $(f, 0)$, since the literal $\neg e$ of ω_3 evaluates to 0 and the propositional variables a and b are still unassigned. The clause ω_3 can be satisfied by setting $(a, 1)$ or $(b, 1)$. Hence, the implied values common to BCP$(a, 1)$ and BCP$(b, 1)$ will be the inferred assignments under $(f, 0)$. In other words, the set $\{\text{BCP}(a=1 \text{ and } f=0) \cap \text{BCP}(b=1 \text{ and } f=0)\}$ will yield the inferred assignments. In this case $(m, 0)$ is the common assignment, and hence we can derive the clause $(f \vee \neg m)$ from Theorem 2 and add it to the existing CNF database. Now let us see how this logic reasoning in clauses is in exact compliance with the logic reasoning utilized for extended backward implications in the gate level circuit netlist. As described in Section 2.1.3, extended backward implications are computed by considering the *target gate* and the *unjustified output specified gates* in the implication list of the target gate. The unassigned fanins of the *unjustified output specified gate* are set to a logic value v one by one, such that the unjustified gate becomes justified and the resulting common set of implications become the new implications of the target gate. Recalling the example in Section 2.1.3, f is the target gate, e is the unjustified output specified gate, a and b are the unassigned fanins of the gate e, and the consequent logic reasoning on unjustified output specified gate e yields the implication $f = 0 \rightarrow m = 0$ or the clause $(f \vee \neg m)$. Thus the logic reasoning in the CNF formula and the circuit netlist yields the same clause. However, the set of clauses obtained through extended backward implications is a subset of all the clauses obtained by Theorem 2. The reason is that we work on the circuit netlist, and while computing extended backward implications we consider only the *unjustified output specified gates* in the implication list of the *target gate*. Hence, not all the affected clauses $\omega \in \Phi$ are checked for satisfiability under the current assignment. For example, under the assignment $(f, 0)$, Theorem 2 will cause the clauses ω_3, ω_{15}, ω_{16}, ω_{17}, ω_{18} and ω_{21} to be checked for satisfiability, whereas extended backward implications only checks the clause ω_3 for satisfiability. This reduces the computational complexity, although at a cost of loosing some highly non-trivial implications which could otherwise be obtained by the implementation of Theorem 2 on to the CNF formula. □

5 Implementation Algorithm

The flow of our algorithm is described below.

Algorithm:

Step 1. Generate the CNF formula for the miter circuit under verification.
Step 2. Compute the direct and indirect implications for each of the nodes in a *levelized* fashion (from the primary inputs towards the primary outputs).
Step 3. (a) Convert the indirect implications obtained in Step 2 into two-literal clauses. (b) Append these new clauses to the CNF database. (c) Add the nodes identified as constants, as *unit* clauses.
Step 4. If more than *n* % of the mitered XOR outputs have been identified as constant 0's, go to Step 7, else go to Step 5.
Step 5. For each gate *N*, compute its extended backward implications.
Step 6. Convert the extended backward implications obtained in Step 5 into binary clauses, and append them to the existing CNF formula.
Step 7. Give the modified CNF formula to a SAT solver for processing.
Step 8. Stop.

6 Experimental Results

The algorithm presented in Section 5 was implemented in C++ in a preprocessing engine called *IMP2C* (Implications to Clauses). *IMP2C* builds the Implication Graph for the miter circuit under verification, and formulates the two-literal clauses corresponding to indirect and extended backward implications learned. The experiments were run on a Pentium 4, 1.8-GHz machine, with 512 MB of RAM and Mandrake Linux 7.2 as the operating system. The efficacy of our technique is corroborated by using the *large* and *difficult* ISCAS'85 benchmark circuits [Brglez 85], the ISCAS'89 full-scan circuits [Brglez 89], the ITC'99 full-scan circuits [Corno 00] and some cascaded ITC'99 benchmarks. Two different types of miter circuits were verified for equivalence: *circuit_eqv* represents an equivalence checking circuit model where two identical copies of the same circuit are mitered, *circuit_opt* represents mitering of the original copy of the circuit and an optimized version (obtained by using *Synopsys* tool). For both miter circuits, we OR all the mitered outputs, and ask the SAT solver to satisfy the *OR gate* output to logic 1.

6.1 Comparison with state-of-the-art SAT solvers

We used two different state-of-the-art SAT solvers, namely, BerkMin561 [Goldberg 02b] and Siege_v4 [Ryan 03] to check the satisfiability of each of the Combinational Equivalence Checking (CEC) instances. All the experiments with BerkMin were run using strategy 1 which is known to be a special strategy for equivalence checking. Also, we added *levels of variables* to the CNF formula given to BerkMin, such that offered the choice of variables to be used in decision making, the variables closer to the inputs will be preferred [Goldberg 03]. Experiments were also run with ZChaff 2001.2.17 [ZhangL 01], but the results have not been reported since for most of the instances ZChaff [ZhangL 01] was found to be 2-10 times slower than BerkMin [Goldberg 02b] and Siege [Ryan 03].

In Table 2, for each miter circuit, we report the execution time taken by our preprocessing engine IMP2C, the time taken by the SAT solver alone, and the time taken by IMP2C + SAT solver together. We also report the speedup ratio of IMP2C + SAT solver over SAT solver alone. The results are reported with $n = 25\%$ in Step 4 of the implementation algorithm described in Section 5. However, it should be noted that our preprocessor can be tuned to handle any threshold given at run time.

Miter Circuit	IMP2C (secs)	Siege (secs)	IMP2C + Siege (secs)	Speed-up (col 3 / col 4)	BerkMin (secs)	IMP2C + BerkMin (secs)	Speedup (col 6 / col 7)
c3540_eqv	0.94	22.21	0.97	**22.89**	1.33	0.97	1.37
c5315_eqv	0.68	12.04	0.69	17.44	1.60	0.80	2.00
c7552_eqv	1.71	34.52	1.76	19.61	11.88	2.10	5.65
c3540_opt	0.82	30.34	1.16	26.15	1.45	1.15	1.26
c5315_opt	16.24	16.23	16.25	0.99	2.32	16.29	0.14
c7552_opt	30.47	39.61	30.48	1.29	12.34	30.49	0.40
s38417_fs_eqv	62.77	336.02	88.60	3.79	163.22	112.24	1.45
s38584.1_fs_eqv	240.02	131.76	267.47	0.49	150.22	300.19	0.50
s35932_fs_eqv	66.28	97.27	81.58	1.19	134.16	69.76	1.92
b14_eqv	26.05	417.13	27.67	15.07	112.67	30.72	3.66
b14_1_eqv	14.50	284.20	15.77	18.02	39.20	17.44	2.24
b15_opt_eqv	57.72	73.90	69.38	1.06	104.67	89.78	1.16
b17_opt_eqv	245.02	458.04	316.08	1.44	846.84	344.78	2.45
b18_opt_eqv	2132.50	5780.29	2557.29	2.26	>14400.0	2497.48	**5.76**
b20_1_eqv	27.88	396.96	36.73	10.80	145.88	36.58	3.98
b21_1_eqv	29.61	427.63	37.37	11.44	142.67	39.75	3.58
b22_1_opt_eqv	43.11	507.00	61.33	8.26	299.72	57.54	5.20
cascade_1*	380.72	2785.30	440.26	6.32	1231.76	520.67	2.36
cascade_2	124.32	1892.65	140.62	13.45	502.62	120.45	4.17
cascade_3	2584.20	6217.78	2947.12	2.10	>14400.0	3078.53	4.67
cascade_4	428.67	8842.92	470.23	18.80	1623.67	404.43	4.01
cascade_5	102.23	1654.41	129.65	12.76	496.42	107.78	4.60

*cascade_1 = b17_opt_b14_eqv, cascade_2 = b14_b22_1_opt_eqv, cascade_3 = b18_opt_b15_opt_eqv, cascade_4=b17_opt_b15_opt_eqv, cascade_5 = b20_1_b21_1_eqv

Table 2: Results with SAT solver alone and (*IMP2C + SAT solver*)

From Table 2, we see that considerable speedup is achieved for almost all the instances. In some cases, once the implication relations are computed, the SAT solver can determine the formula to be unsatisfiable almost immediately. For instance, in the miter circuits *c7552_eqv* and *c3540_opt*, without any added clauses, Siege spent *34.52 seconds* and *30.34 seconds*, respectively. When we augment the CNF formula with the global implication relations (derived by IMP2C), the complexity of the CNF instance is notably reduced, with IMP2C + Siege taking (*1.71 + 0.05*) *1.76 seconds* and (*0.82 + 0.34*) *1.16 seconds*, respectively. Note that the SAT solver Siege takes only a fraction of a second. For the instance *b18_opt_eqv*, BerkMin alone could not

finish even after *4 hours* or *14,400 seconds*, but after IMP2C clauses are added the instance is solved in (*2,132.50 + 364.98*) *2,497.48 seconds*; the time taken by BerkMin being *364.98 seconds* and the time taken by IMP2C being *2,132.50 seconds.*

Unlike Siege, BerkMin uses special equivalence checking strategy (strategy 1 and levels of variables), and hence for most of the cases the speedups with BerkMin are somewhat smaller than with Siege. For some of the relatively easier CEC instances (*e.g.*, *c5315_opt*, *c7552_opt*, *s38584.1_fs_eqv*), the preprocessing due to indirect and extended backward implications was a bit of an overhead, and thus not much speedup was obtained. However, it should be noted that after our preprocessing has been applied, the time taken by the SAT solver alone reduces significantly for all the instances. This suggests that the clauses added are extremely powerful and cause considerable search space pruning, reducing the SAT instance complexity immensely. Overall, the results for IMP2C + SAT solver are very encouraging, with the maximum speedup for IMP2C + BerkMin being *5.76×* in *b18_opt_eqv* and for IMP2C + Siege being *22.89×* in *c3540_eqv*. Since considerable speedup is achieved with each of the SAT solvers, our approach is orthogonal to the two SAT solvers used.

The ISCAS'85 benchmark *c6288* is a 16-bit multiplier circuit and its corresponding miter instances are known to be very difficult for SAT solvers. Hence, we have treated *c6288_eqv* and *c6288_opt* instances separately. Table 3 shows the performance of each of the SAT solvers for these instances without and with our preprocessing technique. The results as we see are very encouraging. For example, the Siege SAT solver could solve the *c6288_eqv* and *c6288_opt* miters in *4,852.30 seconds* and *5,214.50 seconds*, respectively. However, after our preprocessing technique (IMP2C) was applied, the two instances were quickly solved by the SAT solver in less than one-tenth of the second; the preprocessing time being *0.35 seconds* and *3.88 seconds* for *c6288_eqv* and *c6288_opt*, respectively. BerkMin with its special equivalence checking strategy and levels of variables also gave very good results.

Miter Circuit	IMP2C (secs)	Siege (secs)	IMP2C + Siege (secs)	Speedup (col 3 / col 4)	BerkMin (secs)	IMP2C + BerkMin (secs)	Speedup (col 6 / col 7)
c6288_eqv	0.35	4,852.30	0.36	13,478.60	145.30	0.39	372.56
c6288_opt	3.88	5,214.50	3.90	1,337.05	177.73	3.92	45.33

Table 3: Results for c6288 with SAT solver alone and IMP2C+SAT solver

In Table 4 we give the number of clauses in the original CNF formula, the time taken by our preprocessing technique (IMP2C), the number of clauses added using IMP2C, and finally the ratio of added clauses to original clauses. We observe from Table 4 that as the size of the circuit (# original clauses) increases, the time for IMP2C increases in proportion, since many circuit nodes need to be processed for static implications. Also, some circuit structures are such that there are a lot of implication relations among the nodes and hence IMP2C takes a long time. One such case is *b15_opt_eqv* for which IMP2C deduced more than twice the number of

clauses that were in the original CNF formula. It must be noted that even though many clauses were added, we achieved noteworthy speedup for almost all cases, suggesting that the clauses deduced were extremely helpful in pruning the SAT solver search space. Overall, the ratio of added clauses to original clauses varied from *0.29* for *s38417_eqv* to *2.37* for *b15_eqv*, with the mean being *0.95.*

Miter Circuit	Original #Clauses	IMP2C (secs)	Added #Clauses (IMP2C)	Added #Clauses/ Original #Clauses
c3540_eqv	9,462	0.94	4,116	0.44
c5315_eqv	15,743	0.68	6,123	0.39
c6288_eqv	14,788	0.35	6,956	0.47
c7552_eqv	20,504	1.71	13,080	0.64
c3540_opt	9,262	0.82	3,780	0.40
c5315_opt	14,151	16.24	7,261	0.51
c6288_opt	14,719	3.88	8,700	0.59
c7552_opt	20,111	30.47	11,800	0.59
s38417_eqv	127,580	62.77	38,029	0.29
s38584.1_eqv	123,052	240.02	51,894	0.42
s35932_fs_eqv	111,200	66.28	39,977	0.35
b14_eqv	60,661	26.05	75,980	1.25
b14_1_eqv	42,203	14.50	45,968	1.09
b15_opt_eqv	51,329	57.72	121,928	2.37
b17_opt_eqv	165,189	245.02	361,882	2.19
b18_opt_eqv	486,717	2,132.50	866,832	1.78
b20_1_eqv	87,582	27.88	73,379	0.83
b21_1_eqv	87,760	29.61	80,483	0.93
b22_1_opt_eqv	103,173	43.11	84,789	0.82
cascade_1	226,143	380.72	405,874	1.79
cascade_2	164,043	124.32	220,174	1.34
cascade_3	537,289	2,584.20	1,006,745	1.87
cascade_4	217,054	428.67	462,183	2.12
cascade_5	175,985	102.23	168,143	0.95

Table 4: Number of original and added clauses for different CEC instances

In Table 5, we compare our results with those obtained with C-SAT-Jnode [Lu 03a], P_EQ + Berkmin [Novikov 03] and Hypre [Bacchus 03] for ISCAS'85 *ckt*_eqv versions. In [Lu 03a], the authors introduced *incremental learn-from-conflict* strategy. Their algorithm divides the problem at hand into unsatisfiable sub-problems and adds the conflict-induced clauses resulting from solving these sub-problems to the original CNF formula. In [Bacchus 03], the authors utilize hyper binary resolution and equality reduction to simplify the CNF formula. Their tool Hypre can either prove the unsatisfiability of the given CNF formula or yield a simplified CNF formula with fewer variables and clauses. The *ckt_eqv* versions in Table 5 were all proved unsatisfiable by Hypre. According to Table 5, our results are mostly on the same

order of computational effort, and in a few cases better than [Lu 03a, Bacchus 03]. In [Novikov 03], the author gave a theoretical framework for deducing multi-literal relationships. However, a restricted version of the technique was implemented, which deduced only pure and equivalent literals. In our approach, in addition to deducing pure and equivalent literals we deduce non-trivial implication relationships as well. These relationships, when added to the CNF database, are very helpful in reducing the SAT instance complexity as has been shown in the experimental results.

Miter Circuit	C-SAT-Jnode [Lu 03a] (secs)	P_EQ + Berkmin [Novikov 03]† (secs)	Hypre [Bacchus 03] (secs)	IMP2C + BerkMin (secs)	IMP2C + Siege (secs)
c1355_eqv	0.07	0.05	0.15	0.06	0.07
c1908_eqv	0.11	0.27	0.14	0.07	0.08
c2670_eqv	0.13	0.17	0.13	0.42	0.30
c3540_eqv	1.21	0.83	0.86	0.96	0.97
c5315_eqv	0.28	0.61	0.68	0.88	0.69
c6288_eqv	4.14	0.17	0.98	0.35	0.36
c7552_eqv	1.62	0.87	1.48	2.00	1.76

† Expts. were run on Pentium-III, 700 MHz with 640 MB RAM [Novikov 03]

Table 5: Comparison of IMP2C with [*Lu 03a*], [*Novikov 03*] *and* [*Bacchus 03*] *for ISCAS' 85 ckt_eqv benchmarks*

6.2 Comparison with the preprocessor Hypre

We performed another set of experiments to show that the clauses obtained using our preprocessing technique are more powerful and non-trivial than those obtained using Hypre [Bacchus 03]. The results substantiating this are shown in Table 6. The *ckt_opt* CNF instances shown here could not be proved unsatisfiable by Hypre alone and the resulting simplified CNF formula was given to Siege for processing. The CNF instance *c7552_1_opt* used here is much more optimized than *c7552_opt* used in Table 2; *c7552_opt* was proved unsatisfiable by Hypre alone and did not yield any simplified formula. For the *ckt_eqv* versions in Table 6, Hypre did not yield any simplified formula and proved the unsatisfiability immediately. Therefore, for these instances in columns 2 and 5 we take the Siege time to be *0.0 seconds*. For each of the circuits we give the time taken by Hyper + Siege together, followed by the total time taken by IMP2C + Siege. In column 4 we give the speedup of IMP2C + Siege relative to Hyper + Siege. It was observed that when the augmented CNF formula (with IMP2C clauses) was given to Hypre for preprocessing and the resulting simplified formula to Siege, the time to prove unsatisfiability further reduced. The results for this are given in column 5. In column 6 we give the speedup of (IMP2C + Hypre + Siege) over (Hypre + Siege). For a few of the larger instances *s38417_fs_eqv*, *s38584.1_fs_eqv*, *s35932_fs_eqv*, *b18_opt_eqv*, *b17_opt_eqv*, *cascade_1*, *cascade_3* and *cascade_5*, Hypre gave segmentation fault since it has a limit on the number of literals it can handle in a clause (maximum clause length allowed being approximately 1000). The results for these instances have therefore not been reported with Hypre.

We observe from Table 6 that for most of the instances our technique is more superior than Hypre. As evident from column 4, we consistently get a speed up of close to *2×*, with the maximum speedup being *5.76×* for the instance *b15_opt_eqv*. It

has been shown earlier by means of examples (see Section 3.3) that the non-trivial cl auses obtained using our approach cannot be obtained using Hypre. We achieved speedups ranging from *1.36×* to *5.76×.* For example, with *b14_eqv* Hypre + Siege spent *74.2 seconds* whereas IMP2C + Siege spent *27.67 seconds* to prove the unsatisfiability, yielding a speedup of *2.68×.* For six cases, the speedup in column 6 is slightly greater than that in column 4; the reason is that in our approach (IMP2C + Siege), we just augment the original CNF formula with non-trivial two-literal clauses, but do not involve in any equality reduction as is done in Hypre (see Section 1). On the other hand, the CNF formula in column 5 after preprocessing with IMP2C undergoes equality reduction by Hypre, thereby yielding a much simplified and smaller CNF instance. As a result, slightly better execution times are obtained in column 5 than in column 3. One prominent instance where IMP2C + Siege outperforms IMP2C + Hypre + Siege is *b15_opt_eqv*. For this instance, IMP2C + Siege took just (*57.72 + 11.66*) *69.38 seconds*, while IMP2C + Hypre + Siege spent (*57.72 + 196.78 + 0.0*) *254.29 seconds* for preprocessing. Here the preprocessing due to Hypre was an overhead and did not help in reducing the overall execution time.

Miter Circuit	Hypre + Siege (secs)	IMP2C + Siege (secs)	Speedup (col 2 / col 3)	IMP2C+ Hypre+ Siege (secs)	Speedup (col 2 / col 5)
c3540_opt	1.58	1.16	1.36	1.44	1.09
c7552_1_opt	8.68	12.24	0.70	7.24	1.20
s38417_fs_eqv	SF*	88.60	——	SF	——
s38584.1_fs_eqv	SF	267.47	——	SF	——
s35932_fs_eqv	SF	81.58	——	SF	——
b14_eqv	74.20	27.67	2.68	30.07	**2.46**
b14_1_eqv	24.50	15.77	1.55	15.97	1.53
b15_opt_eqv	400.12	69.38	**5.76**	254.29	1.57
b17_opt_eqv	SF	316.08	——	SF	——
b18_opt_eqv	SF	2557.29	——	SF	——
b21_1_eqv	71.29	37.37	1.90	34.57	2.06
b20_1_eqv	65.33	36.73	1.77	33.60	1.94
b22_1_opt_eqv	105.81	61.33	1.72	46.36	2.28
cascade_1	SF	440.26	——	SF	——
cascade_2	186.23	140.62	1.32	134.54	1.38
cascade_3	SF	2,947.12	——	SF	——
cascade_4	SF	470.23	——	SF	——
cascade_5	265.67	129.65	2.04	117.41	2.26

*SF – Segmentation Fault

Table 6: Comparison of IMP2C with Hypre [*Bacchus 03*]

6.3 Comparison with the preprocessor NIVER

To further underpin the superiority of our technique we conducted some experiments with a very recent preprocessor NIVER [Subbarayan 04]. The results are shown in Table 7. As can be observed from column 4 of Table 7, except for a few easy cases,

IMP2C + Siege outperforms NIVER + Siege by a large margin with the maximum speedup being more than *40,000×* for the instance *c6288_eqv*. Similarly, with the other benchmarks such as *b14_eqv*, *b21_1_eqv* and *b20_1_eqv* we achieve more than *50×* speedup. Careful observations of the results show that NIVER may not be very effective for equivalence checking (EC) instances. In fact, NIVER destroys the structure of the EC problem in such a manner that it even becomes difficult for the SAT solver to solve it. This is very much evident from the instances *c6288_eqv*, *c6288_opt* and *b18_opt_eqv* which were all solved in less than *6,000 seconds* by Siege alone. But after preprocessing with NIVER they all took more than *4 hours* or *14,400 seconds.*

Another set of experiments were conducted to see the effect of the combined preprocessing of IMP2C + NIVER on the performance of the SAT solver. This time the results were very notable with IMP2C + NIVER + Siege, yielding the best speedup over NIVER + Siege. Comparing column 3 and column 5, we see that IMP2C + NIVER + Siege takes the least time for almost all the cases. The reason is that our preprocessing tool adds a large number of binary clauses which induces signal correlations into the original CNF formula in terms of equivalent literals, unit clauses and other implication relationships among the CNF variables. As a result, variable elimination by NIVER no longer destroys the problem structure because many relationships have already been deduced by IMP2C. Rather, variable elimination through NIVER reduces the CNF size with the learned relationships from IMP2C intact, and thus helps the SAT solver to prove unsatisfiability quickly.

Miter Circuit	NIVER + Siege (secs)	IMP2C + Siege (secs)	Speedup (col 2 / col 3)	IMP2C+ NIVER+ Siege (secs)	Speedup (col 2 / col 5)
c3540_opt	12.56	1.16	10.82	1.19	10.55
c7552_1_opt	13.54	12.24	1.10	10.58	1.24
c6288_eqv	>14,400.00	0.36	**>40,000.00**	0.38	**>37,894.70**
c6288_opt	>14,400.00	3.90	>3,962.30	3.91	>3,682.86
s38417_fs_eqv	57.61	88.62	0.65	62.87	0.91
s38584.1_fs_eqv	47.76	267.47	0.18	240.23	0.19
s35932_fs_eqv	46.54	81.58	0.57	66.37	0.70
b14_eqv	1,828.67	27.67	66.08	26.14	69.95
b14_1_eqv	468.52	15.77	29.70	14.50	32.31
b15_opt_eqv	70.64	69.38	1.01	63.25	1.11
b17_opt_eqv	472.34	316.08	1.49	260.15	1.81
b18_opt_eqv	>14,400.00	2557.29	>5.63	2142.87	>6.71
b21_1_eqv	2,132.51	37.37	57.06	29.52	72.23
b20_1_eqv	3,150.42	36.73	85.77	31.48	100.07
b22_1_opt_eqv	1,688.45	61.33	27.53	43.51	38.80

Table 7: Comparison of IMP2C with NIVER [*Subbarayan 04*]

6.4 Comparison with the propositional formula checker HeerHugo

In Section 1 we mentioned a propositional formula checker HeerHugo [Groote 00] which adopted a branch/merge rule to prove the satisfiability/unsatisfiability of CNF instances. While the Boolean reasoning utilized in HeerHugo is similar to the extended backward implications that we use, there are several differences. We observed in Section 4.3 that the logic reasoning utilized in extended backward implications is in immediate compliance with Theorem 2, and consequently the set of clauses obtained through extended backward implications is a proper subset of clauses obtained by Theorem 2. Hence, we will use Theorem 2 as the reference to show the differences between extended backward implications and HeerHugo.

Our Boolean reasoning in Theorem 2 involves assigning a logic value 0 or 1 to each of the propositional variables one at a time, and finding the common set of implications required to satisfy the clauses affected under the given assignment. These affected clauses are those where at least one literal evaluates to 0 and where several literals are unassigned under the given assignment. For example, if $\omega = (l_1 \vee l_2 \vee l_3)$ and we make an assignment $x = 0$ such that the literal l_1 evaluates to 0, then the common set of implications obtained by setting $l_2 = 1$ and $l_3 = 1$ independently, will result in binary clauses with x as one of the literals. Here l_2 and l_3 will be pertaining to different propositional variables. Considering the branch/merge rule of level 1 in HeerHugo (see Section 1), after the assignment $x = 0$ is made, another assignment is made on a propositional variable y, no matter even if it appears or does not appear in the clause ω. The common set of implications obtained by setting $y = 0$ and $y = 1$ independently under the assignment $x = 0$ yields binary clauses with x as one of the literals. Note that in HeerHugo the intersection is under the same variable y, whereas in our technique it is under different variables (pertaining to l_2 and l_3 here). The clauses deduced using our technique might require application of the branch/merge rule at higher levels (greater than 2 or 3), and hence the computational complexity of HeerHugo will increase exponentially, since the length of proof to refute the CNF formula Φ in HeerHugo is $O(m^{h+1})$, where m is the total number of clauses and h is the maximum branch/merge level required for proof refutation (please refer to [Groote 00] for details). In our case, if we analyze Theorem 2, the worst case complexity is $O(n.m.p)$, where n is the total number of propositional variables, m is the total number of clauses, and p is the maximum number of literals in a clause in Φ. This is because Theorem 2 operates on each of the n propositional variables in Φ. When any of these propositional variables is assigned to a logic value v, where $v \in \{0, 1\}$ the number of clauses that can get affected in the worst case are m. For every affected clause we need to set each of its unassigned literals to a logic value v, and in the worst case the number of unassigned literals in each of the affected clauses would be $p - 1$ (since at least 1 literal of an affected clause evaluates to 0). Thus the worst case complexity would be $O(n.m.p)$. However, from Observation 2 since we perform our Boolean learning only on a selected set of clauses/variables, based on domain knowledge (the circuit netlist, and not part of the CNF formula), the complexity is significantly lower than $O(n.m.p)$. We believe that the choice of variables on which learning needs to be performed is really important, and selecting these variables judiciously leads to significant gains in terms of inferring new clauses as well as the preprocessor complexity. We use the circuit knowledge to select such variables and

try to deduce non-trivial relations among variables that considerably simplify the equivalence checking instance. The complexity of our preprocessing algorithm is far less than that of a brute force approach involving unstructured reasoning on all the variables. In Table 8 we report the performance of the HeerHugo tool (a complete SAT solver) without and with our preprocessing technique for some of the benchmarks.

Miter Circuit	IMP2C (secs.)	HeerHugo (secs.)	IMP2C + HeerHugo (secs.)	Speedup (col 3 / col 4)
c1908_opt	0.20	24.00	1.20 (0.20 + 1.00)	20.00
c2670_eqv	0.35	16.00	1.35 (0.35 + 1.00)	11.85
c3540_eqv	0.94	763.00	1.94 (0.94 + 1.00)	393.29
c3540_opt	18.10	1308.00	60.10 (18.10 + 42.00)	21.76
c5315_eqv	0.68	1902.00	1.68 (0.68 + 1.00)	1132.14
c5315_opt	12.25	1843.00	50.25 (12.25 + 38.00)	36.67
c7552_eqv	1.71	3443.00	2.71 (1.71 + 1.00)	1270.47
c6288_eqv	0.35	1.00	1.35 (0.35 + 1.00)	0.74
c6288_opt	3.88	1.00	4.88 (3.88 + 1.00)	0.20
s38417_fs_eqv	62.77	>14,400.00	67.77 (62.77 + 5.00)	>212.48
s38584.1_fs_eqv	240.02	>14,400.00	246.02 (240.02 + 6.00)	>58.53
s35932_fs_eqv	66.28	>14,400.00	71.28 (66.28 + 5.00)	>202.02
b14_eqv	26.05	>14,400.00	74.05 (26.05 + 48.00)	>194.46
b14_1_eqv	14.50	>14,400.00	19.50 (14.50 + 5.00)	>738.46
b15_opt_eqv	57.72	>14,400.00	69.72 (57.72 + 12.00)	>206.54
b17_opt_eqv	245.02	Memory Out		
b18_opt_eqv	2,132.50	Memory Out		
b22_1_opt_eqv	43.11	>14,400.00	50.11 (43.11 + 7.00)	>287.36

Table 8: Comparison of HeerHugo[*Groote 00*] *with IMP2C + HeerHugo*

In Table 8, we report the time taken by our preprocessing engine IMP2C, the time taken by the propositional formula checker HeerHugo, and the total time taken by IMP2C + HeerHugo. We also give the speedup ratio of IMP2C + HeerHugo over HeerHugo alone. From Table 8, it is evident that using our preprocessing technique IMP2C boosts up the performance of HeerHugo significantly. For example, HeerHugo alone took *1308.0 seconds* to prove the unsatisfiability of the CNF instance *c3540_opt*, whereas after preprocessing with IMP2C the total time reduced to only *60.10 seconds* (with IMP2C taking *18.10 seconds* and HeerHugo taking *42.0 seconds*, respectively). Similarly, for other instances such as *c1908_opt*, *c2670_eqv*, *c3540_eqv*, *c5315_eqv*, *c7552_eqv* etc. the time taken by HeerHugo reduces from several seconds to *1.0 second.* The ITC benchmarks like *b14_eqv*, *b14_1_eqv*, *b15_opt_eqv* etc. could not be solved by HeerHugo even after *4 hours* (*14,400*

seconds), whereas after preprocessing with IMP2C, these instances became easily tractable. Overall, we achieve speedups ranging from *20.0×* for *c1908_opt* to *1270.47×* for *c7552_eqv*, which is very impressive.

7 Conclusion

We presented a new method of augmenting the original CNF formula with static logic implications. Two-literal clauses resulting from indirect and extended backward implications were quickly computed and added to the existing CNF database. These added clauses served as constraints and helped the SAT solver in the search process. Experimental results for combinational circuit equivalence checking showed that irrespective of the state-of-the-art SAT solver used, we achieved more than one order of magnitude speedup for most of the instances, with the actual speedup ranging up to 22.89×. Comparison with the propositional formula checker HeerHugo [Groote 00] and the recently developed preprocessing techniques such as Hypre [Bacchus 03] and NIVER [Subbarayan 04] showed that our technique exploited the circuit structure very effectively, and significantly reduced the SAT instance complexity achieving orders of magnitude speedup over these methods.

Acknowledgements

We are thankful to the anonymous reviewers for their useful suggestions and comments to improve the quality of the paper.

References

[Bacchus 02] F. Bacchus, "Enhancing Davis Putnam with Extended Binary Clause Recording," In *Proceedings of National Conference on Artificial intelligence* (*AAAI-2002*), August 2002, pp. 613-619.

[Bacchus 03] F. Bacchus and J. Winter, "Effective Preprocessing with Hyper-Resolution and Equality Reduction," In *Lectures notes in Computer Science*, *Theory and Applications of Satisfiability Testing: 6th International Conference*, *SAT 2003*, Volume 2919 / 2004, pp. 341-355.

[Biere 99] A. Biere, A. Cimatti, E. Clarke and Y. Zhu, "Symbolic Model Checking Without BDDs," In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS*) *Conference*, March 1999, pp. 193-207.

[Brglez 85] F. Brglez and H. Fujiwara, "A Neural Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran," In *Proceedings of International Symposium on Circuits and Systems (ISCAS) Conference*, June 1985, pp. 663-698.

[Brglez 89] F. Brglez, D. Bryan and K. Kozminski, "Combinational Problems of Sequential Benchmark Circuits," In *Proceedings of International Symposium on Circuits and Systems* (*ISCAS*) *Conference*, June 1989, pp. 1929-1934.

[Corno 00] F. Corno, M. Sonza Reorda and G. Squillero "RT-Level ITC 99 Benchmarks and First ATPG Results," In *IEEE Design and Test of Computers*, July-August 2000, pp. 44-53.

[Davis 60] M. Davis, H. Putnam "A Computing Procedure for Quantification Theory," In *Journal of ACM*, *Volume 7*, *Issue 3*, July 1960, pp. 201-215.

[Davis 62] M. Davis, G. Longemann and D. Loveland "Machine Program for Theorem Proving," *Communications of the ACM*, Vol. 5, 1962, pp. 394-397.

[Goldberg 02a] E. Goldberg and Y. Novikov, "Berkmin: A Fast and Robust SAT Solver," In *Proceedings of Design*, *Automation and Test in Europe Conference* (*DATE*), March 2002, pp. 142-149.

[Goldberg 02b] E. Goldberg and Y. Novikov, BerkMin561, 2002
http://eigold.tripod.com/BerkMin

[Goldberg 01] E. Goldberg, M. Prasad, R. K. Brayton "Using SAT for Combinational Equivalence Checking," In *Proceedings of Design*, *Automation and Test in Europe Conference* (*DATE*), 2001, pp. 114 -121.

[Goldberg 03] E.Goldberg, Y. Novikov. "Equivalence Checking of Dissimilar Circuits," In *Proceedings of International Workshop on Logic and Synthesis*, May 28-30, 2003.

[Groote 00] J.F. Groote, and J.P. Warners, "The propositional formula checker HeerHugo," In *Journal of Automated Reasoning*, Vol. 24, Nos. 1-2 (February 2000).
http://www.win.tue.nl/~jfg/heerhugo.html

[Gupta 03] A. Gupta, M. Ganai, C. W. Yang and P. Ashar, "Learning From BDDs in SAT-based Bounded Model Checking," In *Proceedings of ACM/IEEE Design Automation Conference* (*DAC*), June 2003, pp. 824-829.

[Kuehlmann 01] A. Kuehlmann, M.K. Ganai and V. Paruthi, "Circuit-Based Boolean Reasoning," In *Proceedings of IEEE/ACM Design Automation Conference* (*DAC*), June 2001, pp. 232-237.

[Kunz 92] W. Kunz and D.K. Pradhan, "Recursive Learning: An Attractive Alternative to the Decision Tree for the Test Generation in Digital Circuits," In *Proceedings of International Test Conference* (*ITC*), September 1992, pp. 816-825.

[Kunz 93] W. Kunz, "HANNIBAL: An Efficient Tool for Logic Verification Based on Recursive Learning," In *Proceedings of IEEE/ACM International Conference on Computer Aided Design* (*ICCAD*), November 1993, pp. 538-543.

[Larrabee 92] T. Larrabee, "Test Pattern Generation using Boolean Satisfiability," In *IEEE Transactions on Computer Aided Design*, Vol. 11, January 1992, pp. 4-15

[Li 00] C. Min Li, "Integrating Equivalency Reasoning into Davis-Putnam Procedure," In *Proceedings of National Conference of Artificial Intelligence* (*AAAI-2000*), July 2000, pp. 291-296.

[Lu 03a] F. Lu, Li-C. Wang, K-T. Cheng and R. C-Y Huang, "A Circuit SAT Solver with Signal Correlation Guided Learning," In *Proceedings of Design*, *Automation and Test in Europe Conference* (*DATE*), March 2003, pp. 892-897.

[Lu 03b] F. Lu, Li-C. Wang, K-T. Cheng, J. Moondanos and Z. Hanna, " A Signal Correlation Guided ATPG Solver and its Applications for Solving Difficult Industrial Cases," In *Proceedings of ACM/IEEE Design Automation Conference* (*DAC*), June 2003, pp. 436-441.

[Lynce 03] I. Lynce and J.P. Marques-Silva, "Probing-Based Preprocessing Techniques for Propositional Satisfiability," In *15th IEEE International Conference on Tools with Artificial Intelligence*, November 2003, pp. 105-110.

[Moskewicz 01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang and S. Malik, "Chaff: Engineering an Efficient SAT Solver," In *Proçeedings of ACM/IEEE Design Automation Conference* (*DAC*), June 2001, pp. 530-535.

[Novikov 03] Y. Novikov, "Local Search for Boolean Relations on the Basis of Unit Propagation," In *Proceedings of Design*, *Automation and Test in Europe Conference* (*DATE*), March 2003, pp. 810 -815.

[Ryan 03] L. Ryan, Siege v4, 2003 http://www.cs.sfu.ca/~loryan/personal, pertinent manuscript available at http://www.cs.sfu.ca/~mitchell/papers/ryan-thesis.ps.

[Subbarayan 04] Sathiamoorthy Subbarayan and Dhiraj K Pradhan, "NiVER: Non Increasing Variable Elimination Resolution for preprocessing SAT instances," In *Proceedings of The Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT '04)*, May 2004

[Schulz 88] M. H. Schulz, E. Trischler and T. M. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System," In *IEEE Transactions on Computer Aided Design*, Vol. 7, January 1988, pp. 126-137.

[Sheeran 00] M. Sheeran and G. Stalmarck, "A Tutorial on Stalmarck's Proof Procedure for Propositional Logic Propositional Proof," In *Formal Methods In System Design*, 16(1), January 2000, pp. 23-58.

[Silva 99a] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," In *IEEE Transaction on Computers*, Vol. 48, May 1999, pp. 506-521.

[Silva 99b] J. P. Marques Silva and L. Guerra E Silva, "Solving Satisfiability in Combinational Circuits using Backtrack Search and Recursive Learning," In *Proceedings of XII Symposium on Integrated Circuits and System Design*, October 1999, pp. 192-195.

[Silva 99c] J. P. Marques-Silva and T. Glass," Combinational Equivalence Checking using Satisfiability and Recursive Learning," In *Proceedings of Design*, *Automation and Test in Europe Conference* (*DATE*), March 1999, pp. 145-149.

[Stalmarck] G. Stalmarck, "System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated from the Boolean Formula," United States Patent Number 5,276,897.

[Stephan 96] P. Stephan, R.K. Brayton and A. L. Sangiovanni Vincentelli, "Combinational Test Generation using Satisfiability," In *IEEE Transactions on Computer Aided Design*, Vol. 15, September 1996, pp. 1167-1176.

[Tseitin 68] G.S. Tseitin, "On the Complexity of Derivation in Propositional Calculus," In *Studies in Constructive Mathematics and Mathematical Logic*, Part 2, 1968, pp. 115-125. Reprinted in J. Siekmann, and G. Wrightson, eds., Automation of Reasoning, Vol. 2, Springer-Verlag, 1983, pp. 466-483.

[Van Gelder 93] A. Van Gelder and Y.K. Tsuji, "Satisfiability Testing with More Reasoning and Less Guessing," In *Second DIMACS Implementation Challenge*, American Mathematical Society, editors D.S. Johnson and M. A. Trick, 1993.

[Zabih 88] R. Zabih and D. McAllester, "A Rearrangement Search Strategy forDetermining Propositional Satisfiability," In *Proceedings of National Conference on Artificial intelligence* (*AAAI-1988*) , pp. 155-160

[ZhangH 97] H. Zhang, "SATO: An Efficient Propositional Prover," In *Proceedings of International Conference on Automated Deduction*, vol. 1249, *LNAI*, July 1997, pp. 272-275.

[ZhangL 01] L. Zhang, C. Madigan, M. Moskewicz and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver," Proceedings of International Conference on Computer Aided Design (*ICCAD*), November 2001, pp. 279-285.

[Zhao 97] J. Zhao, M. Rudnick and J. Patel, "Static Logic Implication with Application to Fast Redundancy Identification," In *Proceedings of VLSI Test Symposium* (*VTS*), April 1997, pp. 288-293.

[Zhao 01] J. Zhao, J. A. Newquist and J. Patel, "A Graph Traversal Based Framework for Sequential Logic Implication with an Application to C-cycle Redundancy Identification," In *Proceedings of VLSI Design Conference*, January 2001, pp. 163-169.