*Journal of Universal Computer Science, vol. 10, no. 12 (2004), 1629-1654*
*submitted: 26/10/03, accepted: 8/12/04, appeared: 28/12/04 © J.UCS*

# A Signal Correlation Guided Circuit-SAT Solver

**Feng Lu, Li-C. Wang, Kwang-Ting (Tim) Cheng**
(Department of Electrical and Computer Engineering,
University of California, Santa Barbara, California, U.S.A.
lufeng|licwang|timcheng@ece.ucsb.edu)

**John Moondanos, Ziyad Hanna**
(Intel Corporation
john.moondanos|ziyad.hanna@intel.com)

**Abstract:** We propose two heuristics, implicit learning and explicit learning, that utilize circuit topological information and signal correlations to derive conflict clauses that could efficiently prune the search space for solving circuit-based SAT problem instances. We implemented a circuit-SAT solver SC-C-SAT based on the proposed heuristics and the concepts used in other state-of-the-art SAT solvers. For solving unsatisfiable circuit examples and for solving difficult circuit-based problems at Intel, our solver is able to achieve speedup of one order of magnitude over other state-of-the-art SAT solvers that do not use the heuristics.

**Key Words:** Boolean Satisfiability, Boolean Equivalence Checking, ATPG
**Category:** B.7.2

## 1 Introduction

Boolean Satisfiability (SAT) has attracted tremendous research effort in recent years, resulting in the development of various efficient SAT solver packages. Popular SAT solvers are designed based upon the Conjunctive Normal Form (CNF) [Zhang 1997, Zhang et al. 2001, Moskewicz et al. 2001, Marques-Silva and Sakallah 1999]. For applications in computer-aided design automation of integrated circuits (CAD), applying SAT to solve a circuit-oriented problem often requires transformation of the circuit gate-level netlist into its corresponding CNF format [Tseitin 1968, Larrabee 1992]. In this circuit-to-CNF transformation, the topological ordering among the internal signals can be obscured. All signals become (input) *variables* in the CNF format.

For solving circuit-oriented problem instances, circuit structural information has proved to be very useful. The authors in [Tafertshofer et al. 1997] developed a structural graph model called an implication graph for efficient implication and learning in SAT. Methods were also provided in [Silva et al. 1999, Silva et al. 2003] to utilize structural information in SAT algorithms, which required minor modifications to the existing SAT algorithms. The authors in [Gupta et al. 2001] implemented a circuit-based SAT-solver that used structural information to identify unobservable gates and to remove the clauses for those gates. The work in [Kuehlmann et al. 2001] represented Boolean circuits in terms of 2-input AND gates and inverters. Based on this circuit model a circuit SAT solver could be integrated with BDD sweeping [Kuehlmann and Krohm 1997].

The authors in [Ganai et al. 2002] developed a circuit-based SAT solver that adopted the techniques used in the CNF-based SAT solver zChaff [Moskewicz et al. 2001], e.g., the watched literal technique for efficient implication. In [Thiffault et al. 2004], techniques such as watched literals used in CNF-based SAT solvers were adopted in the SAT solver for non-clausal formulas.

The work in [Ostrowski et al. 2002] tried to recover the structural information from CNF formulas, and utilized the structural information to eliminate clauses and variables.

The relation between the number of backtracks in DPLL [Davis et al. 1962] SAT-solving procedure and the number of paths in a BDD [Bryant 1986] of the same function was studied in [Reda et al. 2002] where BDD-variable ordering heuristics were used to derive a CNF-variable decision ordering. The work in [Novikov 2003] exploited variable observability by branching on CNF variables and by analyzing the resulting binary values for other variables to derive new implications. The authors in [Cabodi et al. 2003] combined BDD and SAT in the application of Bounded Model Checking [Biere et al. 1999]. In their method, conflict clauses derived from BDD-based approximate traversals were used to prune the search space of SAT-based Bounded Model Checking. Theoretical results about circuit-based SAT algorithms were presented in [Broering et al. 2003].

In this paper, we present a circuit-based SAT solver that utilizes circuit structural information. This paper is an extension of our previous work in [Lu et al. 2003]. The major difference between our approach and other approaches discussed above is in how the circuit structural information is utilized. Our approach includes three new ideas:

1. Our heuristic relies on identifying signal correlations before applying SAT. A group of signals $s_1, s_2, \ldots, s_i$ (where $i > 1$) are said to be possibly correlated if their values satisfy a certain Boolean function $f(s_1, s_2, \ldots, s_i)$ during random simulation, e.g., the values of $s_1$ and $s_2$ satisfy $s_1 = s_2$ during random simulation. In this paper, we only consider three types of signal correlations: equivalence correlation, inverted equivalence correlation, and constant correlation, since they are easy to extract from the results of random simulation. Two signals $s_1$ and $s_2$ have an equivalence correlation (inverted equivalence correlation) if and only if the values of the two signals satisfy $s_1 = s_2$ ($s_1 = \overline{s_2}$) during random simulation. We use $s_1 \sim s_2$ to denote an equivalence correlation between two signals $s_1$ and $s_2$, and use $s_1 \sim s_2'$ to denote an inverted equivalence correlation. If $s_2$ is replaced by constant 0 or 1 in the notation, then that is a constant correlation.

2. When making a decision in SAT solving, it might be more effective to select a variable and assign it a value that is more likely to cause a conflict, so that the conflict could be detected and recorded earlier. Therefore, if we know in advance

how circuit signals are possibly correlated, then that information can be used to guide the decision making in the solver for early conflict detection. We call this procedure that uses signal correlations to affect decision making *implicit learning*.

3. When solving a large complex circuit SAT problem, it might be more efficient to apply incremental solving before solving the problem of the entire circuit. That is, we can first try to solve some subproblems in order to learn useful information, e.g., conflict clauses. This information can then be used to guide the search and prune the search space when solving the original problem. The subproblems can be solved by following the circuit topological structure from inputs to outputs, so that the learned information in solving a smaller subproblem can be used in the solving of a larger subproblem containing the smaller one. This procedure that learns from solving subproblems is called *explicit learning*. The subproblems can be generated from signal correlations, e.g., if $s_1 \sim s_2$, two subproblems "$(s_1 = 0$ and $s_2 = 1)$" and "$(s_1 = 1$ and $s_2 = 0)$," which are likely to be unsatisfiable, can be generated and treated as the subproblems to be solved.

Based on the three ideas above, we implemented a circuit-based SAT solver SC-C-SAT with the two types of the *signal correlation* (SC) learning: implicit learning and explicit learning. For some test cases, SC-C-SAT can perform better than zChaff, BerkMin561 and siege_v4, tools that use no SC learning heuristics. In contrast, our baseline solver without using the SC heuristics does not necessarily outperform these state-of-the-art SAT solvers.

The rest of the paper is organized as follows. In Section 2, we briefly introduce the DPLL SAT algorithm, explain the key ideas in SC learning, and present the *implicit* and *explicit* learning heuristics. In Section 3, we describe the implementation details of our solver. Section 4 describes the SC heuristics used in the experiments. In Section 5 we present experimental results and summarize our findings. In Section 6, we focus on specific experimental results obtained by applying our solver to difficult combinational equivalence problems at Intel. Section 7 concludes the paper.

## 2 The DPLL Algorithm and Our Signal-Correlation Learning Heuristic

Given a finite set of variables, $V$, over the set of Boolean values $\mathbf{B} \in \{0,1\}$, a *literal*, $l$ or $\bar{l}$ is an instance of a variable $v$ or its complement $\neg v$, respectively where $v \in V$. A *clause* $c_i$ is a disjunction of literals $(l_1 \vee l_2 \vee \ldots \vee l_n)$. A formula $f$, is a conjunction of clauses $c_1 \wedge c_2 \wedge \ldots \wedge c_n$. A clause is a set of literals, and a formula is a set of clauses. An assignment $A$ *satisfies* a formula $f$ if $f(A) = 1$. In a Boolean Satisfiability (SAT) problem, a formula $f$ is given and the problem is to find an assignment $A$ to satisfy $f$ or prove that no such assignment exists.

Current complete SAT solvers are almost exclusively based on the DPLL search algorithm proposed in [Davis et al. 1962]. Algorithm 2.1 describes the basic DPLL algorithm.

**Algorithm 2.1:** DPLL()

```
while $(true)$
  do
    if $(!decide())$
      then return $(SATISFIABLE)$
    while $(!bcp())$
      do
        if $(!resolveConflict())$
          then return $(UNSATSFIABLE)$
```

In the algorithm, function *decide*() selects an unassigned variable and assigns it a value. This variable assignment is referred to as a *decision*. If no unassigned variable exists, *decide*() will return *false* which means a solution has been found. Otherwise *decide*() will return *true*. A decision level is associated with each decision. The first decision has decision level 1, and the decision level increases by one for each new decision.

The purpose of *bcp*(), which performs Boolean constraint propagation (BCP), is to identify any variable assignments required by the current variable state for satisfying the formula *f*. In order to satisfy *f*, every clause of it has to be satisfied. Therefore, if a clause has only one unassigned literal and all the other literals are assigned 0, then the unassigned literal must be assigned value 1 in order to satisfy *f* (a clause in this state is said to be unit). This mandatory assignment is referred to as an *implication*. In *bcp*(), BCP is performed transitively, until there are no more implications (in which case *bcp*() returns *true*) or a *conflict* is produced (in which case *bcp*() returns *false*). A conflict occurs when a variable is assigned with both 1 and 0 by implication.

The purpose of *resolveConflict*() is to analyze the reason that causes the conflict (the reason can be recorded as a conflict clause), and try to backtrack to a previous decision level to resolve the conflict. If the conflict could be resolved, the *resolveConflict*() returns *true*. Otherwise, it returns *false*.

In modern SAT solver, one of the key concepts is *conflict-driven learning*. *Conflict-driven learning* is to analyze the reason that causes a conflict and record the reason as a conflict clause to prevent the algorithm from entering the same search space. How to derive conflict clauses that could efficiently prune the search space has been a hot research topic since the introduction of conflict-driven learning.

In this paper, we propose methods that use circuit structural information and signal correlations to guide the SAT solving procedure, so that learned conflict clauses can be used to efficiently prune the search space for circuit SAT problems.

Consider the circuit in Figure 1 where shaded area B contains shaded area A and area C contains shaded area B. Suppose we want to solve a circuit SAT problem with the output objective $c = 1$. When we apply a SAT solver to prove that $c = 1$, potentially the search space for the solver is the entire circuit. Now suppose we can identify, in advance, two internal signals $a$ and $b$, such that $a = 1$ and $b = 0$ individually are very unlikely to happen when random inputs are supplied to the circuit. Then, we can divide

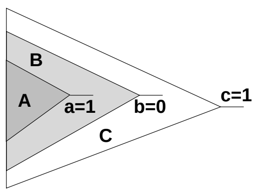

**Figure 1:** *An example for incremental SAT solving*

the original problem into three subproblems: (1) solving $a = 1$, (2) solving $b = 0$, and then (3) solving $c = 1$.

Since $a = 1$ is unlikely to happen, when a SAT-solver makes decisions to satisfy $a = 1$, it is likely to generate conflicts. As a result, conflict-driven information can be learned. In a SAT solver, this information is stored in the learned clauses (or learned gates), each of which represents a functional sub-space containing no solution. ¿From another point of view, each learned clause specifies a constraint on a set of circuit signals that has to be satisfied.

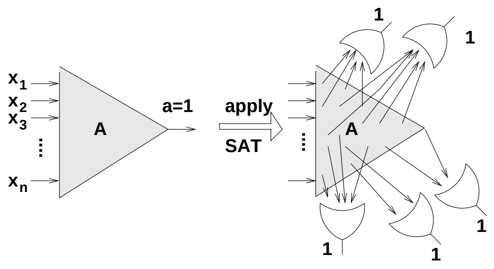

**Figure 2:** *Example learned gates accumulated by solving $a = 1$*

If we assume that solving $a = 1$ is done only based upon the *cone of influence* headed by the signal $a$ (the shaded area A in Figure 1) then the learned clauses will be based upon the signals contained in the area $A$ only. Figure 2 illustrates the results of applying SAT for solving $a = 1$. Regardless of whether the problem is satisfiable or not, a set of learned clauses can be collected. For illustration, in Figure 2 they are represented as the *learned OR gates* whose outputs are 1.

As the solver finishes solving $a = 1$ and starts solving $b = 0$, all the learned information regarding the circuit area $A$ can be used to help solving $b = 0$. In addition, if $a = 1$ is indeed unsatisfiable, then signal $a$ can be assigned 0 when the solver is solving $b = 0$. Similarly, learned information from solving $a = 1$ and $b = 0$ can be reused to help solving $c = 1$.

Intuitively, solving the three subproblems could be much faster than solving the original problem. This is because when solving $b = 0$, hopefully fewer (or no) decisions are required to go into area $A$. Hence, the search space is more restricted within the portion of area $B$ that is not part of area $A$. Similarly, solving $c = 1$ requires most decision making to be done only within the portion of area $C$ that is not part of area $B$. Moreover, the learned clauses accumulated by solving $a = 1$ could be shorter than the conflict clauses accumulated by solving $b = 0$ directly, because the former are based upon the signals in area A only. Similarly, the learned clauses accumulated during the solving of $b = 0$ could be shorter than the conflict clauses accumulated by solving $c = 1$ directly. Conceptually, this strategy allows us to solve a complex problem *incrementally*.

We make two observations: (1) The incremental process suggests that we can guide a solver to solve a sequence of pre-selected subproblems following their topological order. (2) The selection of the subproblems such as $a = 1$ and $b = 0$ should be those most likely to be unsatisfiable. Intuitively, solving a likely unsatisfiable subproblem instance can accumulate learning information more effectively.

If few or no conflicts arise in solving $a = 1$ and solving $b = 0$, then there may be no much information to be learned from solving these two problems. In this case, the above strategy may incur overhead. Usually, this may indicate the ineffectiveness of the method used to guess that $a = 1$ and $b = 0$ are unlikely to happen. Moreover, if solving $c = 1$ does not depend much on signals $a$ and $b$, then the above incremental strategy cannot be effective either. For example, if $c$ is the output signal of a 2-input AND gate with two inputs as a function $g$ and its complement $\overline{g}$, then $c = 1$ can be directly proved to be unsatisfiable without knowing the actual function $g$. To reduce the potential overhead, one can use the signal correlation information such as $a = 1$ and $b = 0$ implicitly rather than explicitly as described above. In an implicit approach, the assignments $a = 1$ and $b = 0$ would be tried before the assignments $a = 0$ and $b = 1$ when the SAT algorithm is required to make a decision to assign values to these two signals.

Intuitively, the above incremental strategy would not be effective for solving a circuit SAT problem whose input is given in CNF form. This is because by treating the CNF form as a 2-level OR-AND circuit structure, the topological ordering among the signals is lost. With a 2-level structure, the incremental strategy has little room to proceed. In the example discussed above, both $a$ and $b$ become primary inputs in the 2-level OR-AND CNF circuit. Then, the ordering for solving the subproblems may become solving $b = 0$ followed by solving $a = 1$.

Another key issue is how to identify in advance the subproblems that are most likely to be unsatisfiable. One straightforward approach to identify signal correlations is based on simulation with random input assignments.

## 2.1 Identifying Signal Correlations

Let $\{s_1, \ldots, s_n\}$ be $n$ signals on a given circuit. We denote $s_i \sim s_j$ for $i \neq j$ if $s_i = s_j$ is true during one run of random simulation (a possible equivalence correlation identified through the simulation). Similarly, $s_i \sim s'_j$ denotes the case where $s_i = \overline{s_j}$ is true during one run of random simulation. Moreover, we also include $s_i \sim 0$ (and $s_i \sim 1$) as a signal correlating to the logic constant 0 (and 1).

Suppose that we have identified a signal correlation between $s_i$ and $s_j$ as $s_i \sim s_j$. Then, intuitively assigning $s_i = 1$ and $s_j = 0$ (or vice versa) will likely cause the SAT process to produce a conflict. Similarly, if we have $s_i \sim 0$, assigning $s_i = 1$ will likely cause a conflict.

In Algorithm 2.2, we demonstrate a simple procedure to compute the set of equivalence and inverted equivalence correlations based on random simulation. Suppose we define the relation $s_i \leftrightarrow s_j$ as $s_i \sim s_j \vee s_i \sim s'_j$ for two signals $s_i$ and $s_j$, i.e., $s_i \leftrightarrow s_j$ means that signals $s_i$ and $s_j$ have equivalence or inverted equivalence correlation. It is not difficult to see that the relation $\leftrightarrow$ is an equivalence relation. i.e., $s_i \leftrightarrow s_i$, $s_i \leftrightarrow s_j \Rightarrow s_j \leftrightarrow s_i$, and $s_i \leftrightarrow s_j \wedge s_j \leftrightarrow s_k \Rightarrow s_i \leftrightarrow s_k$. This equivalence relation can be used to partition signals into equivalence classes.

**Algorithm 2.2:** RANDOM SIMULATION (*Circuit*)

```
comment: $C$ is the initial equivalence class.
$i \leftarrow 0$; $count \leftarrow 0$; $C \leftarrow$ the set of all signals; $S \leftarrow \{C\}$;
while ($i < 4$)
  do {
    $S' \leftarrow \emptyset$;
    Produce 32 random input assignments;
    Perform parallel logic simulation [Abramovici et al. 1990];
    for each equivalence class $P$ of $S$
      do {
        $N \leftarrow$ Compute New Equivalence Class($Circuit$, $P$, $count$);
        $S' \leftarrow S' \cup N$;
      }
    if ($S \neq S'$)
      then $i \leftarrow 0$;
      else $i \leftarrow i + 1$;
    $count \leftarrow count + 1$;
  }
return ($S$)
```

In the algorithm, $S'$ is used to store the equivalence classes derived by the previous simulation iterations, and an *equivalence class* is a subset of signals mutually having the equivalence relationship. Variable *count* is used to count the number of iterations. When computing new equivalence classes, *count* is used to indicate whether the while-loop is in the first iteration. The usage of *count* is shown in Algorithm 2.3, which will be explained in more detail below.

During each iteration, 32 random input assignments are grouped together using a word (32 bits) in parallel logic simulation [Abramovici et al. 1990]. If repeating the simulation step several times (such as four times) does not lead to identifying additional equivalence classes, then the simulation stops, and the equivalence classes are reported.

We note that deriving new equivalence classes from a previous equivalence class based on the current simulation results can be achieved efficiently with a hash table. Algorithm 2.3 illustrates this procedure.

**Algorithm 2.3:** COMPUTE NEW EQUIVALENCE CLASS (*Circuit*, *P*, *count*)

```
comment: $P$ is the previous equivalence class.
comment: $count$ is the iteration count as in Algorithm 2.2.
$N \leftarrow \emptyset$;
if ($count = 0$)
  then for each signal $s$ of $P$
    if (LOOKUP($h$, $K(s)$, $V$))
      then $V \leftarrow V \cup \{s\}$;
           FLAG($s$) $\leftarrow 1$;
    else if (LOOKUP($h$, $\tilde{}K(s)$, $V$)) $V \leftarrow V \cup \{s\}$;
                                              FLAG($s$) $\leftarrow 0$;
    else $V \leftarrow \{s\}$;
         INSERT($h$, $K(s)$, $V$);
         $N \leftarrow N \cup \{V\}$;
         FLAG($s$) $\leftarrow 1$;
  else for each signal $s$ of $P$
    if (FLAG($s$) $= 1$)
      then $key \leftarrow K(s)$;
      else $key \leftarrow \tilde{}K(s)$;
    if (LOOKUP($h$, $key$, $V$))
      then $V \leftarrow V \cup \{s\}$;
      else $V \leftarrow \{s\}$;
           INSERT($h$, $K(s)$, $V$);
           $N \leftarrow N \cup \{V\}$;
return ($N$)
```

In the algorithm, $h$ is a hash table of which the hash key is a 32-bit word, and the hash value is a set of signals. $V$ is a set of signals and $N$ is the set to store new derived equivalence classes. The simulation results of a signal $s$ in the current iteration are grouped as a 32-bit word denoted as $K(s)$, and $\tilde{}K(s)$ is the complement of $K(s)$. FLAG($s$) is the flag of a signal $s$. Function LOOKUP($h$, $k$, $v$) is to search the hash table $h$ with the key $k$. If $k$ is found in $h$, LOOKUP($h$, $k$, $v$) returns $true$ and the hash value associated with the key $k$ is returned by $v$. Otherwise, LOOKUP($h$, $k$, $v$) returns $false$. Function INSERT($h$, $k$, $v$) inserts the key $k$ and its associated value into the hash table $h$.

When equivalence classes are derived, two signals $s_i$ and $s_j$ in an equivalence class have an equivalence correlation if FLAG($s_i$) = FLAG($s_j$). Otherwise they have an inverted equivalence correlation. Other correlations $s_i \sim 0$, and $s_i \sim 1$ can be identified by checking if $s_i$ always has the value 0 (or 1) during the entire run of the random simulation.

In all experiments presented in this paper, random simulation stops after repeating four times and identifying no additional correlation. In general, setting this number below four would not be effective for identifying signal correlations in all the test cases considered in this work. However, we note that increasing this number to a large number such as ten or twenty does not affect the performance trends reported in this work. For all the run-time results in this paper, simulation times are included (if SC learning is involved) and they are usually much smaller than the actual solver run times.

## 2.2 Implicit Learning by Signal Grouping

The implicit learning strategy utilizes signal correlations to influence the decision variable selection. For example, our SAT solver may employ the VSIDS heuristic in the decision variable selection [Moskewicz et al. 2001]. When VSIDS is combined with implicit learning, the implicit learning has a higher priority in selecting the next decision variable.

With implicit learning, we use signal correlations to group variables in the decision variable selection. For example, suppose that a signal $s_i$ is possibly correlated with signal $s_j$ as $s_i \sim s'_j$. During the solving process, whenever $s_i$ is assigned with a constant value (0 or 1), we immediately make the decision to assign the same value to $s_j$. In other words, we group the two signals together in the solver's value-assignment process, and value assignments are done in such a way that they are most likely to cause a conflict. Algorithm 2.4 shows our signal correlation guided implicit learning.

**Algorithm 2.4:** SELECT DECISION VARIABLE - IMPLICIT LEARNING (*variables*)

```
Suppose $s$ is just being assigned a value $v$ by implication (BCP)
if ($\exists s', s'$ is possibly correlated with $s$) and ($s'$ has not yet been assigned a value)
  then { select $s'$ as the next decision signal
         if (it is equivalence correlation)
           then $s' \leftarrow \overline{v}$
           else $s' \leftarrow v$
  else { use VSIDS (or other heuristics) to select a signal $s'$
         if ($s'$ is possibly correlated with 0 or 1)
           then { $s' \leftarrow 1$ if the correlation is "$s' = 0$", or
                  $s' \leftarrow 0$ otherwise
           else $s' \leftarrow$ value based on VSIDS heuristic;
return ($s'$)
```

When selecting a decision variable, the algorithm first checks the last signal implied by the last decision. If the signal has an unassigned possibly correlated signal, the possibly correlated signal will be selected as the next decision variable. Otherwise, the VSIDS (or other heuristics) is used to select the next decision variable.

## 2.3 Explicit Learning

In explicit learning, a sequence of likely unsatisfiable subproblems are created based upon the signal correlations identified by the random simulation. Internally, the solver tries to solve each subproblem one by one, following their topological order. The learned clauses obtained in solving a subproblem are kept and used in solving other subproblems and the original problem.

When solving each subproblem, we need to decide when to stop the solving. To limit the subproblem solving process, one can count the number of learned gates accumulated. For example, the solver can move on to the next subproblem whenever *i* new learned gates are generated. The default *i* value in SC-C-SAT is set at 10, but the user can specify a different number as input to the solver. We note that the optimal value of this number is problem-dependent. In all experiments shown in this paper, we use the default value 10.

Algorithm 2.5 shows our signal correlation guided explicit learning. In this algorithm, $VG$ is an array of all signals of the circuit, sorted based on the topological order from inputs to outputs. The topological levels of signals in the circuit are computed in a way that the primary inputs are all with level 0, and the level of every other signal is the maximum level of its fanins increased by 1. The signals are then arranged in *VG* in ascending order of their topological levels. $VC(s)$ is an array of signals possibly correlated with the signal $s$. Moreover, $VC(s)$ only stores signals whose indexes in $VG$ are smaller than the index of $s$ in $VG$. $VC(s)$ is also sorted based on the topological order from inputs to outputs.

**Algorithm 2.5:** EXPLICIT LEARNING ()

```
$numvg \leftarrow$ the number of elements of $VG$
for $i \leftarrow 0$ to $(numvg - 1)$
do
    $s \leftarrow VG[i]$
    if $(s \sim 0)$
      then solve the subproblem $(s = 1)$
      else if $(s \sim 1)$
      then solve the subproblem $(s = 0)$
      else
          $numvc \leftarrow$ the number of elements of $VC(s)$
          for $j \leftarrow 0$ to $(numvc - 1)$
          do
              $t \leftarrow VC(s)[j]$
              if $(s \sim t)$
                then
                    solve the subproblem $(s = 0,\ t = 1)$
                    solve the subproblem $(s = 1,\ t = 0)$
                else
                    solve the subproblem $(s = 0,\ t = 0)$
                    solve the subproblem $(s = 1,\ t = 1)$
```

We will use $IND(s)$ to designate the index of signal $s$ in the signal array $VG$. Given two possibly correlated signal pairs $(s_1,\ t_1)$ and $(s_2,\ t_2)$, where $s_1$ is possibly correlated with $t_1$, and $s_2$ is possibly correlated with $t_2$, suppose we have $IND(s_1) > IND(t_1)$ and $IND(s_2) > IND(t_2)$. The subproblem of $(s_1,\ t_1)$ will be solved prior to the subproblem of $(s_2,\ t_2)$ if $(IND(s_1) < IND(s_2)) \vee ((IND(s_1) = IND(s_2)) \wedge (IND(t_1) < IND(t_2)))$. Since the signal array $VG$ is sorted based on the topological order from primary inputs to primary outputs, the subproblems are solved following the same ordering.

## 3 Implementation Details

The input to our solver is in a circuit format (".bench" format [Brglez et al. 1985]). After the circuit is read in, we transform it into a netlist based upon only the 2-input

AND primitive. In the netlist, we allow inverters to be associated with the inputs of AND gates as their attributes.

We apply circuit optimization techniques [Kuehlmann and Krohm 1997] to the netlist to merge structure-equivalent gates. This simple check can be done as the following. Two gates $g_1(i_1, i_2)$ and $g_2(j_1, j_2)$ are merged if the gate input $i_1$ is the same signal as gate input $j_1$ and gate input $i_2$ is the same signal as gate input $j_2$. After the merge, the netlist is modified. Suppose we remove $g_2$. Then, all the gates receiving their inputs from $g_2$ will now receive their inputs from $g_1$. The check is performed following the topological ordering. This feature is not turned on for circuit equivalence verification test cases, which are described as "C-" test cases in Section 5 and Intel circuits in Section 6. This is because some "C-" test cases can be verified simply by the circuit structural optimization just described. If a test case is constructed based on two structurally identical circuits, then applying the structural equivalence gate merging just described above, following the circuit topological order, can result in a circuit test case that is trivial to be verified for equivalence.

The techniques used in zChaff, BerkMin and the SAT solver in [Ganai et al. 2002] are combined to implement our baseline solver. For Boolean constraint propagation, lookup tables are used for fast implications on the AND primitive [Ganai et al. 2002], and watched literal technique is used on conflict clauses [Moskewicz et al. 2001].

For decision variable selection, our baseline solver first tries to select a decision variable from the most recently produced conflict clause that is left to be unsatisfied [Goldberg and Novikov 2002]. If all conflict clauses are satisfied, the solver includes the justification nodes (J-node) for consideration in decision variable selection [Abramovici et al. 1990, Ganai et al. 2002].

In ATPG terminology, a J-node is a gate whose output has received a value, and some of its inputs need further decision(s) to justify the value [Abramovici et al. 1990]. In our implementation, when all the conflict clauses are satisfied, the decisions are then made to satisfy J-nodes of the highest topological order. That is, finding the J-node with the largest index in *VG* and selecting the variable from its inputs, which has the highest VSIDS [Moskewicz et al. 2001] score as the decision variable.

UIP based conflict analysis [Zhang et al. 2001] is adopted in our baseline solver. The conflict clause database of our solver is organized as a stack, and each new conflict clause is added to the top of the stack [Goldberg and Novikov 2002]. Every time when 2048 conflict clauses are added to the conflict clause database, a clause removal procedure is invoked. The conflict clauses obtained in solving a subproblem are kept until they are removed by the clause removal procedure.

The clause removal procedure is similar to the one in [Goldberg and Novikov 2002]. Here the conflict clause database is viewed as a queue where newly-generated conflict clauses are always added to the tail of the queue. For the first 1/16 of the total conflict clauses starting from the head of the queue, clauses with more than 8 literals are considered to be removed based on their activities. For the rest 15/16 of the total conflict clauses, clauses with more than 42 literals are considered to be removed based on their activities.

For every 2048 backtracks, our solver restarts if at least one conflict clause with less than 9 literals is added to the conflict clause database during this period.

## 4 Heuristics for the Experiments

Suppose that two signals have the correlations $s_1 \sim 0$ and $s_2 \sim 1$ and topologically, $s_2$ is on a path from $s_1$ to a primary output. Then, it is possible that $s_1 \sim 0$ is the cause for $s_2 \sim 1$. Hence, when the subproblem $s_1 \sim 0$ is aborted, solving $s_2 \sim 1$ may not provide much useful information. To reduce the overhead of explicit learning, when the subproblem $s_1 \sim 0$ is aborted, the subproblem $s_2 \sim 1$ can be ignored. Based on the above argument, we designed a *cut-based heuristic*: when a subproblem derived from a constant-correlated signal $s$ is aborted, all the subproblems derived from constant-correlated signals which are on a path from $s$ to a primary output are ignored. When the cut-based heuristic is applied, it only affects the signals that possibly correlate to 0 or 1. Pair-wise signal correlations are not affected.

Combining implicit learning, explicit learning and cut-based heuristics, we derive more heuristics to experiment with. These experiments show how the degree of learning affects the performance of the solver for different sets of benchmarks. In our experiments, we apply our solver in six different ways as described below.

- Baseline: This is the baseline solver without using any SC heuristic.
- Implicit: Only implicit learning heuristic is applied to our baseline solver.
- ecut: Explicit learning and cut-based heuristics are applied to our baseline solver.
- eicut: Explicit learning, implicit learning, and cut-based heuristics are applied to our baseline solver.
- e: Only explicit learning heuristic is applied to our baseline solver.
- ei: Explicit learning and implicit learning heuristics are applied to our baseline solver.

## 5 Experimental Results

We use three groups of examples. A "C-" test case is constructed based on a benchmark circuit in the ISCAS85 or ISCAS89 suites [Brglez et al. 1985, Brglez et al. 1989]. If it is an ISCAS89 benchmark, the inputs of flip-flops are treated as primary outputs and the outputs of flip-flops are treated as primary inputs. In other words, an ISCAS89 circuit is converted to a combinational circuit.

In the case denoted as "-.eq" we constructed an equivalence checking circuit model by taking two copies of the same circuit. Each pair of corresponding primary outputs is XORed and all the outputs of the XOR go to an OR gate. The SAT problem is to prove whether the output of the OR gate can become 1, which indicates that the two circuits are not equivalent. In each of our experiments, the output of that OR gate is unsatisfiable. In the case denoted as “-.opt” the two copies are structurally different where one copy is logically optimized with a commercial synthesis tool.

A “V-” example is a satisfiable test case taken from [Velev 2000]. A “P-” example is an unsatisfiable test case taken from [Velev 2000]. The CNF formulas for the ”C-” examples used in the experiments for CNF-based solvers are generated based on the netlist model using 2-input AND gates with inverter attributes as described previously.

For comparison purpose, we conducted experiments with three other SAT solvers, zChaff [Moskewicz et al. 2001], BerkMin561 [Goldberg and Novikov 2002] and the solver siege_v4 [Ryan 2004]. All experiments were run on a Pentium-4 2.4-GHz machine with 1.5 GB RAM under Linux Mandrake 2.4.3.

## 5.1 Results on “C-” test cases

| | | | | SC-C-SAT | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Cases | zChaff | BerkMin | siege_v4 | Baseline | Implicit | ecut | eicut | e | ei |
| c3540.opt | 26 | 2 | 17 | 40 | 3.3 | 0.1 | 0.1 | 0.1 | 0.1 |
| c5315.opt | 84 | 3 | 7 | 9 | 2.3 | 0.1 | 0.1 | 0.1 | 0.1 |
| c7552.opt | 299 | 6.5 | 21 | 34 | 5.6 | 0.5 | 0.6 | 0.7 | 0.7 |
| c3540.eq | 35 | 2.8 | 12 | 23 | 4.3 | 0.1 | 0.1 | 0.1 | 0.1 |
| c5315.eq | 39 | 3.4 | 6 | 6 | 2.2 | 0.1 | 0.1 | 0.1 | 0.1 |
| c7552.eq | 190 | 7 | 17 | 26 | 5.1 | 0.7 | 0.7 | 0.7 | 0.7 |
| s38417.eq | 420 | 116 | 64 | 79 | 69 | 3.7 | 3.9 | 1.8 | 1.8 |
| s38584.eq | 316 | 145 | 61 | 125 | 121 | 36 | 36 | 35 | 35 |
| Total | 1409 | 285.7 | 205 | 342 | 212.8 | 41.3 | 41.6 | 38.6 | 38.6 |
| c6288.eq | * | * | * | * | * | 0.1 | 0.1 | 0.1 | 0.1 |

*Aborted after 1 hour.

**Table 1:** *Results (secs) for UNSAT “C-” test cases*

Table 1 summarizes the results of CPU time on “C-” test cases. We observe that explicit learning (column *e*) is more effective than implicit learning in these test cases. Comparing to the baseline solver, using implicit learning achieves on average a 1.6$\times$ speedup and using explicit learning achieves on average an 8.8$\times$ speedup. Our baseline solver is faster than zChaff, comparable to BerkMin, but is slower than siege_v4. The circuit c6288.eq represents a special case where only explicit learning heuristics can solve it within a limited time. And when the right heuristics (explicit learning) are applied, the run times are only less than a second.

Tables 2 and 3 give the number of decisions and the number of conflict clauses on “C-” test cases respectively. It can be seen that the performance of the solvers are not reflected by the number of decisions or the number of conflict clauses. Consider zChaff, BerkMin, siege_v4 and our baseline solver on these test cases, the number of

| | | | | SC-C-SAT | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Cases | zChaff | BerkMin | siege_v4 | Baseline | Implicit | ecut | eicut | e | ei |
| c3540.opt | 92102 | 79130 | 79743 | 111783 | 19911 | 3597 | 3597 | 3597 | 3597 |
| c5315.opt | 276510 | 165827 | 97633 | 91926 | 31836 | 5933 | 5933 | 5933 | 5933 |
| c7552.opt | 799650 | 239938 | 185097 | 202630 | 42482 | 12009 | 11948 | 12623 | 12623 |
| c3540.eq | 98691 | 103780 | 65875 | 79531 | 21854 | 3402 | 3402 | 3402 | 3402 |
| c5315.eq | 217414 | 179516 | 84801 | 81570 | 31470 | 6326 | 6326 | 6326 | 6326 |
| c7552.eq | 571394 | 256607 | 176060 | 170403 | 41996 | 14288 | 13895 | 13993 | 13993 |
| s38417.eq | 2090072 | 4359689 | 2750160 | 1413476 | 956124 | 99028 | 87922 | 37852 | 37852 |
| s38584.eq | 5186352 | 4332569 | 3328875 | 2018729 | 1569106 | 84069 | 63823 | 44097 | 44097 |
| Total | 9332185 | 9717056 | 6768244 | 4170048 | 2714779 | 228652 | 96846 | 127823 | 127823 |
| c6288.eq | * | * | * | * | * | 9070 | 9070 | 9070 | 9070 |

*Aborted after 1 hours.

**Table 2:** *Number of decisions for UNSAT "C-" test cases*

| | | | | SC-C-SAT | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Cases | zChaff | BerkMin | siege_v4 | Baseline | Implicit | ecut | eicut | e | ei |
| c3540.opt | 61966 | 13207 | 58533 | 68553 | 13930 | 1532 | 1532 | 1532 | 1532 |
| c5315.opt | 139937 | 14315 | 37301 | 24196 | 8617 | 2751 | 2751 | 2751 | 2751 |
| c7552.opt | 340660 | 22410 | 69714 | 63442 | 16954 | 4067 | 4479 | 4019 | 4019 |
| c3540.eq | 66819 | 14455 | 47037 | 47786 | 15421 | 1509 | 1509 | 1509 | 1509 |
| c5315.eq | 91899 | 15302 | 31517 | 18125 | 8122 | 2893 | 2893 | 2893 | 2893 |
| c7552.eq | 252491 | 23777 | 65678 | 51202 | 15055 | 4153 | 4882 | 3988 | 3988 |
| s38417.eq | 455242 | 57394 | 73142 | 42349 | 32684 | 14940 | 15223 | 14760 | 14760 |
| s38584.eq | 66023 | 19640 | 25928 | 28428 | 22476 | 18143 | 18637 | 18113 | 18113 |
| Total | 1475037 | 180500 | 408850 | 344081 | 133259 | 49988 | 51906 | 49565 | 49565 |
| c6288.eq | * | * | * | * | * | 4454 | 4454 | 4454 | 4454 |

*Aborted after 1 hours.

**Table 3:** *Number of conflict clauses for UNSAT "C-" test cases*

total decisions of BerkMin is the largest but its total number of conflict clauses is the smallest, and for our baseline solver, the total number of decisions and total number of conflict clauses are both smaller than those of siege_v4.

## 5.2 The ordering of explicit learning

In the above experiments, the ordering of the explicit learning follows the topological order of the signals. In this section, we consider not following the topological order in explicit learning. Results are shown in Table 4.

| | The Ordering in Explicit Learning (based on *ei*) | | |
|---|---|---|---|
| Circuit | Topological | Reverse | Random |
| c3540.eq | 0.1 | 2 | 1.3 |
| c5315.eq | 0.1 | 1.1 | 0.7 |
| c7552.eq | 0.7 | 3.3 | 2.7 |
| c6288.eq | 0.1 | * | * |

*Aborted after 1 hour

**Table 4:** *Effects of changing the ordering in explicit learning for SC-C-SAT*

It can be seen that if the topological order is not followed, the effectiveness of explicit learning degrades. As we can observe, random ordering is better than reverse ordering, and both are inferior to topological ordering. One noticeable result is that if we do not follow the topological order, then the solver would not be able to complete the run for c6288.eq.

The results in Table 4 demonstrate the importance of following the topological order in the incremental learning process. For this reason, we suspect that for those "V-" and "P-" examples, where the resulting circuits fed to our solver are in two-level OR-AND structure (directly translated from their CNF forms), the proposed explicit learning may produce inferior results. Without the topological ordering, the effectiveness of the incremental learning is degraded, and the overhead associated with the incremental process can out-weight the potential efficiency gain from explicit learning.

## 5.3 Results on "V-" and "P-" test cases

For the "V-" and "P-" examples, the circuits fed to our solver are 2-level circuits obtained directly from their CNF formulas, where the structural information was lost, so that explicit and implicit learning could not help much.

| | | | | SC-C-SAT | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Cases | zChaff | BerkMin | siege_v4 | Baseline | Implicit | ecut | eicut | e | ei |
| Vliw001 | 356 | 165 | 47 | 73 | 72 | 169 | 128 | 218 | 222 |
| Vliw002 | 336 | 117 | 5 | 69 | 147 | 130 | 262 | 153 | 152 |
| Vliw003 | 381 | 213 | 70 | 16 | 100 | 181 | 216 | 262 | 210 |
| Vliw005 | 1088 | 96 | 37 | 134 | 99 | 191 | 136 | 176 | 158 |
| Vliw006 | 308 | 150 | 41 | 53 | 251 | 186 | 252 | 194 | 206 |
| Vliw008 | 550 | 103 | 45 | 93 | 187 | 115 | 181 | 163 | 214 |
| Vliw011 | 305 | 68 | 7 | 95 | 58 | 152 | 126 | 152 | 172 |
| Total | 3324 | 912 | 252 | 533 | 914 | 1124 | 1301 | 1318 | 1334 |

**Table 5:** *Results (secs) for SAT "V-" test cases*

Table 5 summarizes the results on the "V-" test cases. It is interesting to observe that explicit learning performs worse than implicit learning and implicit learning performs worse than the baseline. We conclude that SC heuristics are not effective for these examples. Hence, the more we depend on the SC learning, the worse the results would be. Our baseline solver is on average five times faster than zChaff, slightly faster than BerkMin, but is two times slower than siege_v4 for these test cases.

Table 6 summarizes the results for "P-" test cases. In this table, no clear trend can be concluded. Our baseline solver achieves a 1.68× average speedup over siege_v4 and 2× average speedup over BerkMin.

| | | | | SC-C-SAT | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Cases | zChaff | BerkMin | siege_v4 | Baseline | Implicit | ecut | eicut | e | ei |
| 3Pipe1 | 62 | 17 | 25 | 10 | 8 | 11 | 11 | 10 | 11 |
| 3Pipe2 | 87 | 28 | 19 | 14 | 10 | 16 | 11 | 16 | 11 |
| 4Pipe1 | 753 | 155 | 102 | 104 | 115 | 102 | 159 | 102 | 162 |
| 4Pipe2 | 2505 | 285 | 229 | 130 | 128 | 134 | 161 | 134 | 161 |
| 4Pipe3 | * | 254 | 266 | 123 | 127 | 155 | 134 | 150 | 157 |
| Total | * | 739 | 641 | 381 | 388 | 418 | 476 | 412 | 502 |

*Aborted after 1 hours.

**Table 6:** *Results (secs) for UNSAT "P-" test cases*

## 6 Solving Hard Equivalence Checking Cases

In this section, we report our experience with difficult test cases from an Intel Pentium III class microprocessor. These test cases come from hard Combinational Equivalence Checking (CEC) problems where HDL specifications are compared against their gate level implementations. The heuristic *eicut* of SC-C-SAT is used for evaluation since it is the default heuristic and also performs well for the "C-" examples.

**Circuit Characteristics:** With advancements in the area of CEC, one does not expect to see many difficult signals in a microprocessor [Kuehlmann and Krohm 1997, Moondanos et al. 2001]. The number of logic levels that can exist within a pipe stage of modern microprocessor designs is limited, and therefore sophisticated CEC techniques prove highly efficient. However with increased market segmentation for microprocessors, our experience has been that a few specific signals can prove particularly hard for certain CEC technologies. Here, we studied the performance of our solver on test cases coming from four units of an Intel microprocessor design. These test cases were derived from combinational equivalence checking of RTL against gate level implementations of signals in these four units that belong to the memory cluster. Circuit $C_a$ performs memory command stream control. Circuit $C_m$ is a memory address generation circuit which contains multiple arithmetic units such as adders and multipliers. Circuit $C_n$ performs memory address translation based on operand type, while circuit $C_o$ implements a read-only interface to memory.

**Why They Are Difficult:** These combinational equivalence problems have proved particularly difficult for BDD-based techniques. Traditional monolithic BDD comparison does not work regardless of complicated variable ordering schemes, including dynamic re-ordering. This is reasonable given the arithmetic logic that is included in these circuits. In addition, divide-and-conquer techniques based on BDDs which incorporate key point matching have also failed on many signals in these circuits. Intuitively, this is due to the fact that these circuits contain many re-convergent signals which create false negatives for cut-point techniques that do not offer false negative elimination. On the other hand, employing cut-point [Kuehlmann and Krohm 1997] based techniques that offer false-negative elimination by employing parametric representations for functions, such as *normalized BBDs* [Moondanos et al. 2001], does not work because of the degree of re-convergence. However, the existence of significant re-convergence is a clear indication that signals in the netlists are highly correlated and this is something that should be exploited to speed up the process of a decision procedure. Also there is no significant structural similarity between the two netlists being compared because one is coming from the RTL compilation process and the other is from the optimized gate-level netlist.

**Results Comparison:** In the figures from Figure 3 to Figure 14, the horizontal axis corresponds to numerical indices that were assigned to the various CEC problems in $C_a, C_m, C_n$ and $C_o$. These indices were ordered according to the run times given by SC-C-SAT. Figures 3, 4, 5 and 6 show the time performance comparison for circuit signals in $C_a, C_m, C_n$ and $C_o$ respectively. For BerkMin561, option "s 1" was adopted which determined the decision-making strategy for equivalence checking problems. A missing point in these figures (resulting in a discontinuous curve) indicates that the particular solver did not finish the run for the test case within the time limit. The run time limit was set at 10000 seconds.

Figure 3 gives the time performance comparison for problems of circuit $C_a$. Our solver could finish any of the problems within 10 seconds. BerkMin561 is faster than Siege_v4 on these problems and could finish any of the problems within 1000 seconds. Some of the problems couldn't be solved by zChaff within 10000 seconds.

Figure 4 shows the time performance comparison for problems of circuit $C_m$. Our solver performs well on most of the problems, and only for a few problems, Siege_v4 is better than our solver. BerkMin561 and Siege_v4 each have problems on which it performs better than the other, but Siege_v4 could finish more signals than BerkMin561.

Figure 5 shows the time performance comparison for problems of circuit $C_n$. On these problems, our solver is about 100 times faster than BerkMin561, the performance of BerkMin561 and Siege_v4 are quite similar, and zChaff is about 100 times slower than BerkMin561.

Figure 6 shows the time performance comparison for problems of circuit $C_o$. On these problems, our solver is the fastest and BerkMin561 is slightly better than Siege_v4.

Figures 7, 8, 9 and 10 present the decision number comparison for circuit signals in $C_a, C_m, C_n$ and $C_o$ respectively. Note that the signal IDs of the horizontal axis of these figures are the same as the signal IDs of Figures 3, 4, 5 and 6 respectively.

Figure 7 shows the decision number comparison for problems of circuit $C_a$. On these problems, the number of decisions is greatly reduced by our solver compared with the other three solvers, zChaff, BerkMin and Siege_v4. The number of decisions of Siege_v4 is smaller than that of BerkMin. For some cases that zChaff could finish in 10000 seconds, the number of decisions of zChaff is smaller than that of BerkMin.

Figure 8 shows the decision number comparison for problems of circuit $C_m$. On most of the problems, the number of decisions of our solver is the smallest. BerkMin and Siege_v4 each have some cases that need fewer decisions than the other. For cases that zChaff could finish in 10000 seconds, the number of decisions of zChaff is close to that of BerkMin.

Figure 9 shows the decision number comparison for problems of circuit $C_n$. On most of the problems, the number of decisions of our solver is at most 1/10th of that of Siege_v4, the number of decisions of Siege_v4 is smaller than that of BerkMin and the number of decisions of BerkMin is smaller than that of zChaff.

Figure 10 shows the decision number comparison for problems of circuit $C_o$. On all of the problems, our solver has the smallest decision number. Siege_v4 has the second smallest decision number. BerkMin has the third one and the decision number of zChaff is the largest.

Figures 11, 12, 13 and 14 show the number of conflict clauses comparison for circuit signals in $C_a$,$C_m$,$C_n$ and $C_o$ respectively. Note that the signal IDs of the horizontal axis of these figures are the same as the signal IDs of Figures 3, 4, 5 and 6 respectively.

Figure 11 shows the conflict clause number comparison for problems of circuit $C_a$. On these problems, the number of conflict clauses is greatly reduced by our solver compared with the other three solvers, zChaff, BerkMin and Siege_v4. The number of conflict clauses of Siege_v4 is smaller than that of BerkMin. The number of conflict clauses of zChaff is the largest.

Figure 12 shows the conflict clause number comparison for problems of circuit $C_m$. On most of the problems, the number of conflict clauses of our solver is the smallest. BerkMin and Siege_v4 each have some cases that generate fewer conflict clauses than the other. zChaff has the largest conflict clause number on most cases.

Figure 13 shows the conflict clause number comparison for problems of circuit $C_n$. On most of the problems, the number of conflict clauses of our solver is at most 1/10th of that of BerkMin, the number of conflict clauses of BerkMin is smaller than that of Siege_v4 and the number of conflict clauses of Siege_v4 is smaller than that of zChaff.

Figure 14 shows the conflict clause number comparison for problems of circuit $C_o$. On all of the problems, our solver has the smallest conflict clause number. BerkMin has the second smallest conflict clause number. Siege_v4 has the third one and the conflict clause number of zChaff is the largest.

The comparisons take into account only those pairs of signals that gave rise to extremely difficult combinational equivalence problems. The experiments were conducted using Pentium III workstations running Red Hat Linux 6.2 with 4 GB of RAM.

¿From the figures one can conclude that the performance of SC-C-SAT (eicut) is at least one order-of-magnitude more efficient than the performance of the other three solvers, in terms of run time, number of decisions, and number of conflict clauses. We also note that SC-C-SAT could finish in total 40% more signals than zChaff and 8% more signals than BerkMin for all the signals shown in these figures.

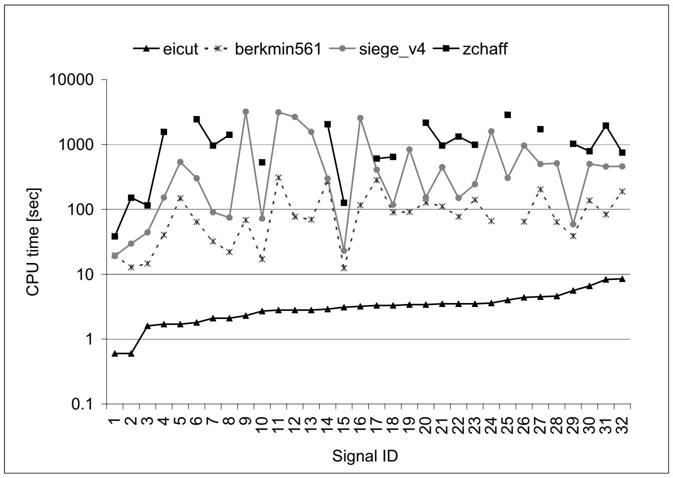

Figure 3: *SC-C-SAT (eicut) vs. zChaff vs. BerkMin vs. Siege_v4 CPU time comparison on hard signals form circuit $C_a$*

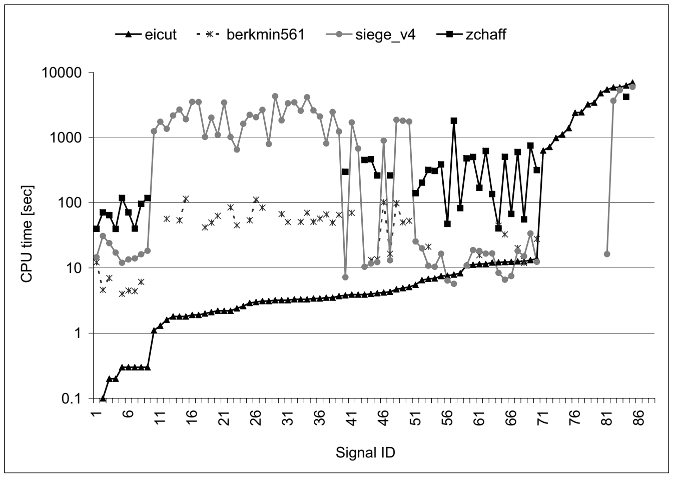

Figure 4: *SC-C-SAT (eicut) vs. zChaff vs. BerkMin vs. Siege_v4 CPU time comparison on hard signals form circuit $C_m$*

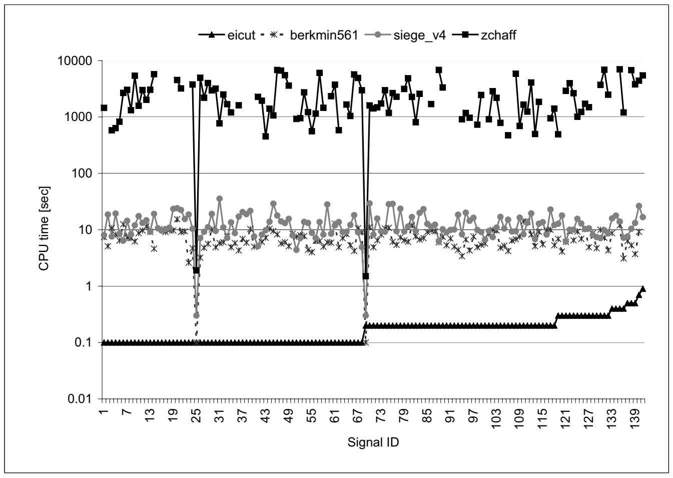

Figure 5: *SC-C-SAT (eicut) vs. zChaff vs. BerkMin vs. Siege_v4 CPU time comparison on hard signals form circuit $C_n$*

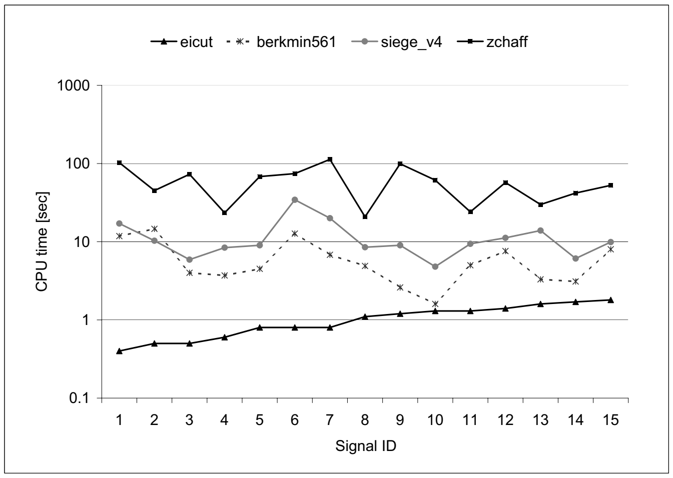

Figure 6: *SC-C-SAT (eicut) vs. zChaff vs. BerkMin vs. Siege_v4 CPU time comparison on hard signals form circuit $C_o$*

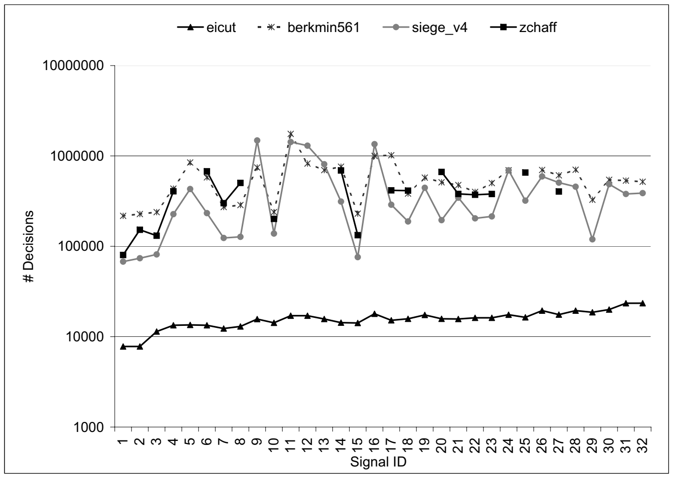

Figure 7: *SC-C-SAT (eicut) vs. zChaff vs. BerkMin vs. Siege_v4 decision number comparison on hard signals form circuit $C_a$*

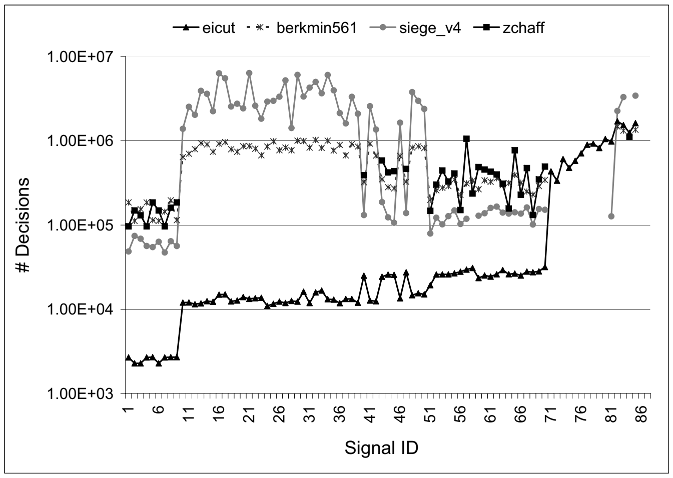

Figure 8: *SC-C-SAT (eicut) vs. zChaff vs. BerkMin vs. Siege_v4 decision number comparison on hard signals form circuit $C_m$*

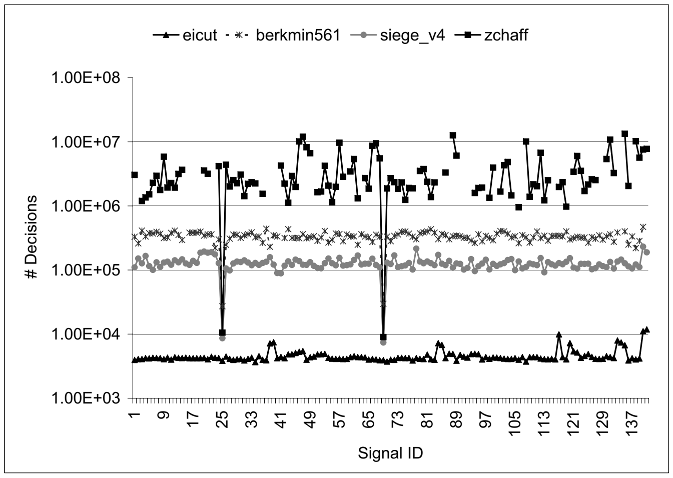

Figure 9: *SC-C-SAT (eicut) vs. zChaff vs. BerkMin vs. Siege_v4 decision number comparison on hard signals form circuit $C_n$*

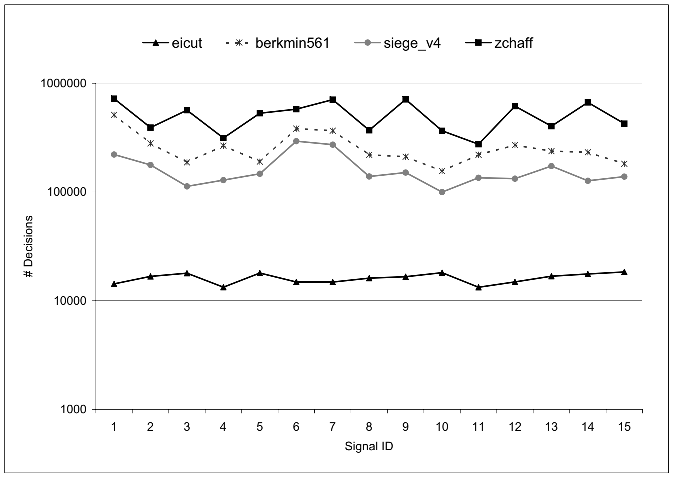

Figure 10: *SC-C-SAT (eicut) vs. zChaff vs. BerkMin vs. Siege_v4 decision number comparison on hard signals form circuit $C_o$*

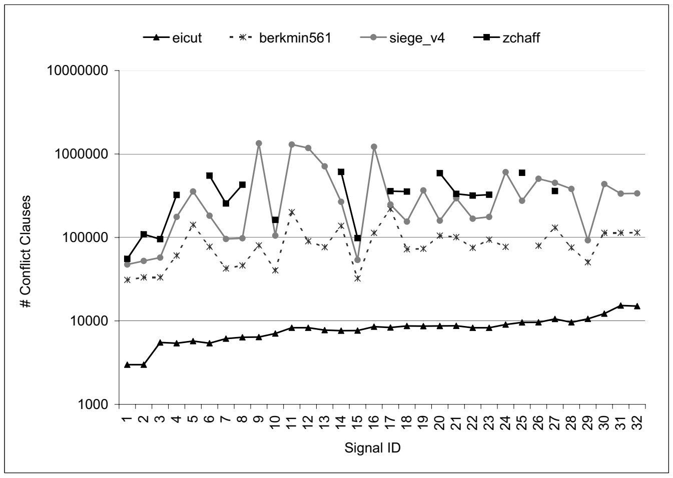

Figure 11: *SC-C-SAT (eicut) vs. zChaff vs. BerkMin vs. Siege_v4 conflict clause number comparison on hard signals form circuit $C_a$*

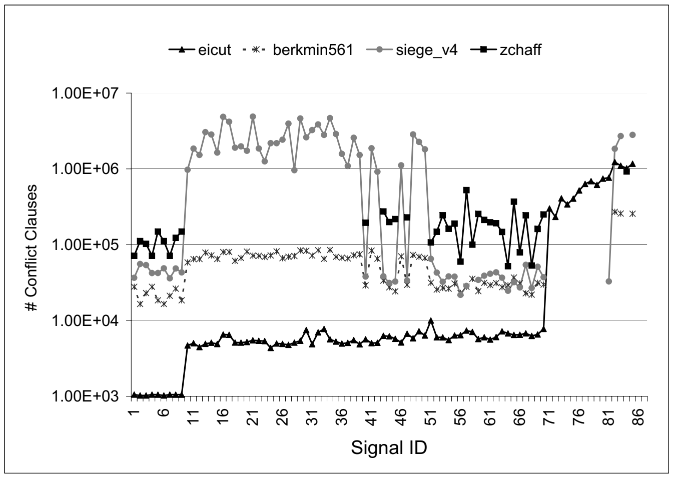

Figure 12: *SC-C-SAT (eicut) vs. zChaff vs. BerkMin vs. Siege_v4 conflict clause number comparison on hard signals form circuit $C_m$*

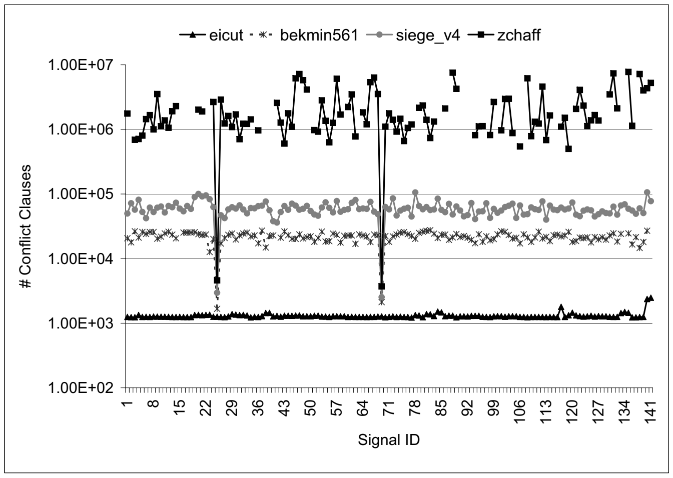

Figure 13: *SC-C-SAT (eicut) vs. zChaff vs. BerkMin vs. Siege_v4 conflict clause number comparison on hard signals form circuit $C_n$*

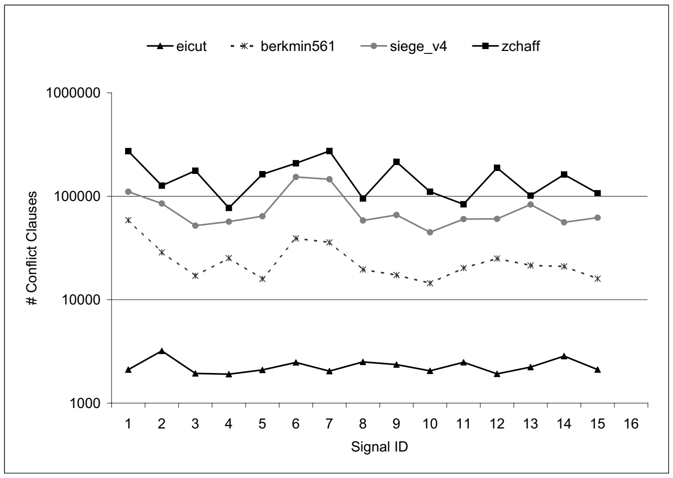

Figure 14: *SC-C-SAT (eicut) vs. zChaff vs. BerkMin vs. Siege_v4 conflict clause number comparison on hard signals form circuit $C_o$*

## 7 Conclusions

This paper described two heuristics, implicit learning and explicit learning that are circuit-based. Our heuristics utilize signal correlations and circuit topological information to improve the performance of a circuit-based SAT solver for circuit-oriented problems. We discussed the strengths and weaknesses of the proposed two heuristics, and compare their performance to other state-of-the-art SAT solvers. Although we do not consider that our SC-C-SAT solver is always superior to zChaff, BerkMin, and seige_v4 when solving all problem instances (especially if the input format is CNF-based), for some examples shown in this paper, our solver is able to take advantage of the circuit structural information and achieve significant performance improvements. SC-C-SAT can be downloaded from our web site http://cadlab.ece.ucsb.edu.

## References

[Abramovici et al. 1990] M. Abramovici, M.A. Breuer, and A.D. Friedman, Chapters 3,5, and 6: Logic Simulation, Fault Simulation, Test Generation, *Digital Systems Testing and Testable Design*, W. H. Freeman, 1990.

[Biere et al. 1999] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking using SAT Procedures instead of BDDs," In *Proc. of the 36th ACM/IEEE Design Automation Conf.*, June 1999, pp. 317-320.

[Brglez et al. 1985] F. Brglez, P. Pownall, R. Hum, "Accelerated ATPG and Fault Grading via Testability Analysis," In *Proc. of the International Conference on Circuit and Systems, 1985, pp. 695-698.*

[Brglez et al. 1989] F. Brglez, D. Bryan, K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits," In *Proc. of ISCAS-89, pp. 1929-1924.*

[Broering et al. 2003] E. Broering, and S.V. Lokam, "Width-Based Algorithms for SAT and CIRCUIT-SAT," In *Proc. of Theory and Applications of Satisfiability Testing,* May 2003, pp. 162-171.

[Bryant 1986] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," In *IEEE Transactions on Computers*, Vol. C-35, No. 8, 1986 , pp. 677-691.

[Cabodi et al. 2003] G. Cabodi, S. Nocco, and S. Quer, "Improving SAT-Based Bounded Model Checking by Means of BDD-Based Approximate Traversals," *Design, Automation, and Test in Europe (DATE '03), March 2003, pp. 898-903.*

[Davis et al. 1962] M. Davis, G. Longeman, and D. Loveland, "A Machine Program for Theorem Proving," *Communications of the ACM,* vol.5, 1962, pp. 394-397.

[Ganai et al. 2002] M.K. Ganai, L. Zhang, P. Ashar, A. Gupta, and S. Malik, "Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver," In *Proc. ACM/IEEE Design Automation Conference,* June 2002, pp. 747-750.

[Goldberg and Novikov 2002] E. Goldberg, Y. Novikov, "BerkMin: A fast and robust Sat_Solver," In *Proc. Design, Automation and Test in Europe,* March 2002, pp. 142-149.

[Gupta et al. 2001] A. Gupta, Z. Yang, and P. Ashar, "Dynamic Detection and Removal of Inactive Clauses in SAT with Application in Image Computation," In *Proc. ACM/IEEE Design Automation Conference,* June 2001, pp. 536-541.

[Kuehlmann and Krohm 1997] A. Kuehlmann and F. Krohm, "Equivalence Checking using Cuts and Heaps," *Proceedings Design Automation Conference, 1997,* pp. 263-268.

[Kuehlmann et al. 2001] A. Kuehlmann, M. Ganai, and V. Paruthi, "Circuit-based Boolean Reasoning," in *Proc. ACM/IEEE Design Automation Conference,* June 2001, pp. 232-237.

[Larrabee 1992] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," In *IEEE Transactions on Computer-Aided Design,* Jan, 1992, pp. 4-15.

[Lu et al. 2003] F. Lu, L.C. Wang, K.T. Cheng, R. Huang, "A circuit SAT solver with signal correlation guided learning," In *Proc. Design, Automation and Test in Europe* 2003, pp. 10892-10897.

[Marques-Silva and Sakallah 1999] J.P. Marques-Silva and K.A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *IEEE Trans on Computers, vol.48* 1999, pp. 506-521.

[Moondanos et al. 2001] J. Moondanos, C. Seger, Z. Hanna and D. Kaiss, "CLEVER: Divide and Conquer Combinational Logic Equivalence VERification with False Negative Elimination," *Proc. Computer-Aided Verification Conference, 2001*, pp. 131-143.

[Moskewicz et al. 2001] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proc. Design Automation Conference* 2001, pp. 530-535.

[Novikov 2003] Y. Novikov, "Local Search for Boolean Relations on the Basis of Unit Propagation," in *Proc. Design, Automation and Test in Europe* 2003, pp. 10810-10815.

[Ostrowski et al. 2002] R. Ostrowski, E. Grgoire, B. Mazure, and L. Sas, "Recovering and Exploiting Structural Knowledge from CNF Formulas," *Principles and Practice of Constraint Programming (CP '02)*, P. Van Hentenryck, ed., LNCS 2470, Springer-Verlag, September 2002, pp. 185-199.

[Reda et al. 2002] S. Reda, R. Drechsler, and A. Orailoglu, "On the Relation Between SAT and BDDs for Equivalence Checking," *International Symposium on Quality of Electronic Design (ISQED '02)*, 2002, pp. 394-399.

[Ryan 2004] L. Ryan, the siege satisfiability solver. http://www.cs.sfu.ca/ loryan/personal/.

[Silva et al. 1999] L. Silva, L. Silveira, and J.M. Silva, "Algorithms for Solving Boolean Satisfiability in Combinational Circuits," in *Proc. Design, Automation and Test in Europe*, 1999, pp. 526-530.

[Silva et al. 2003] L. Silva and J.M. Silva, "Solving Satisfiability in Combinational Circuits," in *IEEE Design and Test of Computers*, July/August, 2003, pp. 16-21.

[Tafertshofer et al. 1997] P. Tafertshofer, A. Ganz, and M. Henftling, "A SAT-Based Implication Engine for Efficient ATPG, Equivalence Checking, and Optimization of Netlists," in *Proc. International Conference on Computer-Aided Design*, 1997, pp. 648-657.

[Thiffault et al. 2004] C. Thiffault, F. Bacchus, and T. Walsh, "Solving Non-clausal Formulas with DPLL search," in *Tenth International Conference on Principles and Practice of Constraint Programming (CP '04)*, 2004, pp. 663-678.

[Tseitin 1968] 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.

[Velev 2000] M.N. Velev, http://www.ece.cmu.edu/˜mvelev Benchmark Suites, October 2000.

[Zhang et al. 2001] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik, "Efficient conflict driven learning in a Boolean satisfiability solver," in *Proc. International Conference on Computer-Aided Design* 2001, pp. 279-285.

[Zhang 1997] H. Zhang, "SATO: An Efficient Propositional Prover," *Proc. of International Conference on Automated Deduction*, Vol 1249, LNAI, 1997, pp. 272-275.