A BDD-Based Satisfiability Infrastructure using the Unate Recursive Paradigm

Priyank Kalla, Zhihong Zeng, Maciej J. Ciesielski and Chilai Huang
Department of Electrical and Computer Engineering
University of Massachusetts Amherst, MA-01003
{pkalla, zzeng, ciesiel}@ecs.umass.edu

Technical Report: TR-CSE-99-6

ABSTRACT

Keywords: Boolean Satisfiability, Unate and Binate Functions.

The Boolean satisfiability (SAT) problem is the problem of finding a solution to the equation $f = 1$, where f is a Boolean formula to be satisfied. The formula f can be represented in Conjunctive Normal Form (CNF), or by using Binary Decision Diagrams (BDDs). Previous BDD-based SAT solutions assume that the conjunction of all the constraints can be represented by a monolithic BDD. However, the resulting BDD is often prohibitively large. Assuming that the individual constraints can be represented using BDDs, but the BDD representing the conjunction of all the constraints cannot be built because of its prohibitive size, how do we find a satisfiability solution? This paper presents a comprehensive infrastructure to efficiently solve the above problem.

Two efficient search algorithms are presented. The first algorithm, called BSAT, is a recursive, backtracking algorithm to search for a SAT solution. The well known *unate recursive paradigm* is exploited to solve the SAT problem. We recursively apply orthonormal expansion on highly binate functions that may eventually lead to cofactors that are unate. Search for satisfiability solutions on these unate cofactors can be efficiently performed. However, because of the exhaustive nature of search, BSAT spends a lot of time searching for solutions that either do not exist, or are hard to find. To overcome this deficiency, we present another algorithm, called INCOMPLETE-SEARCH-USAT (abbreviated IS-USAT).

IS-USAT, is an iterative, non-backtracking algorithm that incorporates an *incomplete* search to find a satisfying solution. The search is incomplete inasmuch as it is restricted to only those regions that have a high likelihood of containing the solution, discarding the rest. Storing a restricted search space in BDDs avoids BDD-size explosion problems while further reducing the time of search. However, since only a part of the search space is explored, feasible SAT solutions that may exist only in unexplored regions cannot be found by this technique. We present a heuristic (based on the unateness properties of functions) that attempts to explore those regions of search space where there is a high chance of finding a solution. While this technique cannot be relied upon to prove unsatisfiability in unequivocal terms, we demonstrate with experiments that it is able to compute feasible SAT solutions quickly for a large number of SAT instances. Using the above SAT engines, we were able to find SAT solutions not only for all ISCAS'85 and MCNC benchmark circuits, but also for a variety of industry standard designs.

While the recursive paradigm has been successfully applied to various applications in logic synthesis, in the context of this paper, we demonstrate its effectiveness in solving the SAT problem efficiently, while controlling the growth of BDD size.

This work has been supported in part by a grant from Avery Design Systems, Inc.

I. INTRODUCTION

The Boolean satisfiability problem (henceforth called SAT) has many direct applications in the electronic design automation (EDA) arena, which includes test pattern generation, timing analysis, logic verification, functional testing, etc. The SAT problem belongs to the class of NP-complete problems, with algorithmic solutions having exponential worst-case complexity [1]. This problem has been widely investigated, and continues to be so because efficient techniques can greatly impact the performance of satisfiability tools.

With respect to applications in VLSI-CAD, most instances of SAT formulations start from an abstract circuit description, for which a given circuit property - typically, value requirements at primary outputs - needs to be validated. The resulting formulation is then mapped onto an instance of SAT, in many cases using Conjunctive Normal Form (CNF) formulae. Classical approaches to SAT are based on variations of the well known Davis and Putnam procedure [2]. Typical versions of the above [3] [4] [5] [6] incorporate a chronological backtrack based search that, at each node in the search tree, selects an assignment and prunes subsequent search by iterative application of the unit clause and pure literal rules [7]. Recent approaches [8] [9], incorporate learning techniques and other conflict analysis procedures with non-chronological backtracks to prune the search space. In spite of these advances, CNF-SAT models and associated algorithms have several drawbacks which prompted us to consider other resources, namely, Reduced Ordered Binary Decision Diagrams (ROBDDs) [10] [11].

If multiple instances of SAT need to be solved, mapping each problem description onto CNF-SAT can represent a large percentage of the total running time [5] [9]. This can be a performance bottleneck for other applications that rely extensively on information provided by SAT tools (*e.g.*, ATPG [5] [6], equivalence checking [12], functional vector generation, etc.). Moreover, the requirement that the constraints be in CNF form leads to formulae with too many clauses [13]. The choice of ROBDDs was thus dictated not only by the issue of inter-operability of our SAT engines with various BDD-based verification/testing frameworks[1], but also because of their compactness and their ease of manipulation.

Let us briefly describe the SAT problem as it appears in the context of our work and analyze the limitations of previous BDD-based solutions. Consider the circuit shown in Fig. 1. Given a set of *output value requirements*, say $\{u = 1, v = 1, w = 1\}$, how do we find an *input assignment* that satisfies these requirements?

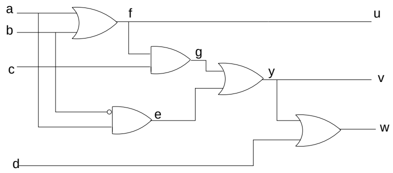

Fig. 1. An Example circuit.

Building the product of the output function requirements in terms of the primary inputs results in all satisfying solutions. Let f represent the product of the outputs. So, $f = u \cdot v \cdot w = ac + bc + a\overline{b}$. It follows that any of the cubes ac, bc, or $a\overline{b}$ provide the satisfying assignments. Similary, if the output value requirements are $\{u = 1, v = 1, w = 0\}$, then $f = u \cdot v \cdot \overline{w} = 0$, which implies that there is no SAT assignment for this instance.

Previous BDD-based SAT approaches [14][13] model the overall constraints (*i.e.* conjunction of u, v, w) using a monolithic BDD. Thus, if the product f is represented by a monolithic ROBDD, selecting any cube as a SAT solution amounts to traversing any path from the root node to the terminal vertex **1** [13] [14][15]. While this simple formulation is appealing because it leverages the BDD manipulation algorithms, it is often the case that constructing and storing the product BDD f for large problems is not feasible. If the product f is large enough that it cannot be stored using a BDD, *how do we find a satisfying assignment without attempting to construct such a prohibitively large product BDD*? Solutions to this problem are the subject of this paper.

Using the techniques of [14] [13] [15] (termed BDD-SAT in this paper), we carried out a few experiments by constructing a BDD for the product of all output functions for the benchmark circuits. Experimental data is provided in Table I. Such a straightforward approach performs well for a wide range of medium sized circuits. However, for large circuits the conjunction operation is either too

[1] The satisfiability engines presented in this paper are integrated within a functional validation tool. It is our desire to automatically generate *functional vectors* at the inputs of a design that would assist in its functional validation. The user provides specific value requirements at the primary outputs for a design. The SAT engines are then used to identify the value assignments to the inputs that satisfy these requirements. The input value assignments thus generated by the SAT tools can be used for functional simulation purposes.

slow (benchmark *C5315*), or it results in a prohibitivley large BDD that cannot be stored (*des, C6288*).

TABLE I
BDD-SAT: SAT USING PRODUCT OF BDDS

Benchmark	#PI	#PO	CPU T	End Product BDD size
C1355	41	35	6 sec	153 nodes
C7522	207	108	40 sec	5561 nodes
C5315	178	123	24 min	21487 nodes
13×13mult	26	26	36 sec	15269 nodes
32-b alu	73	32	76.6 sec	237611 nodes
pair	173	137	2 min	7013 nodes
des	256	245	–	out of memory
C6288	32	32	–	out of memory

While carrying out these experiments, we made the following observation. While the *end product* BDD is generally quite small, the size of the intermediate products can be prohibitively large. For instance, the size of the end product BDD for the benchmark *pair* was just 7,013 nodes. However, the size of the largest intermediate product was 382,493 nodes. This means that the intermediate products contain vast regions of empty solution space.

This issue of BDD-size explosions of intermediate computations was also observed by Hu *et al.* in [16]. They proposed the idea of implicitly conjoined BDDs to reduce the size of intermediate computations by targeting the *functionally dependent variables* [17], a technique similar in flavour to that of partitioned BDDs [18]. The basic idea here was to represent a prohibitively large BDD as a conjunction of small BDDs represented in terms of intermediate variables. While these techniques [17][16][18] can be used to avoid BDD-size blow-ups by using intermediate variables, the SAT solutions are ultimately required in terms of the primary input variables. Resolving these intermediate (dependent) variables in terms of the primary input (independent) variables again leads to BDD-size explosion problems. Thus, in order to prevent BDD-size explosion of intermediate computations, it becomes important to discard these regions of empty solution space from further consideration. This issue motivates the SAT techniques presented in this paper.

This paper presents a comprehensive infrastructure to efficiently solve the Boolean satisfiability problem using BDDs. In particular, two efficient search algorithms are presented. The algorithms exploit characteristics of unate functions in order to intelligently discard regions of the search space from further exploration, thus arresting the growth of BDD size. The first algorithm presented in this paper, called BSAT, is a recursive, backtracking algorithm to find a SAT solution. The well known *unate recursive paradigm* [19] [20] is exploited to solve the SAT problem. We recursively apply orthonormal expansion on highly binate functions that may eventually lead to cofactors that are unate. Search for SAT solutions on the resulting unate cofactors can be efficiently performed. While the recursive paradigm has been successfully applied to various applications in logic synthesis, in the context of this paper, we demonstrate its effectiveness in solving the SAT problem efficiently, while controlling the growth of BDD size. While experiments over a wide range of benchmarks reveal that BSAT successfully arrests the growth of BDD size, because of the (semi-)exhaustive nature of search, it suffers from higher computation times. The second algorithm presented in this paper, called INCOMPLETE-SEARCH-USAT, attempts to overcome this deficiency.

INCOMPLETE-SEARCH-USAT (abbreviated as IS-USAT in the sequel) is an iterative, non-backtracking search algorithm to solve the SAT problem. The search is incomplete inasmuch as it is restricted to those regions that have a *high likelihood* of containing a solution, while discarding the rest. Storing a restricted search space in BDDs avoids BDD-size explosion problems and further reduces the time of search. However, since only a part of the search space is explored, feasible SAT solutions that may exist only in unexplored regions cannot be found by this technique. We present a heuristic (based on the unateness properties of functions) that attempts to explore those regions of search space where there is a high chance of finding a solution. While this technique cannot be relied upon to prove unsatisfiability in unequivocal terms, we demonstrate with experiments that it is able to compute feasible SAT solutions quickly for a large number of SAT instances. Using the above SAT engines, we were able to find SAT solutions not only for all ISCAS'85 and MCNC benchmark circuits, but also for a variety of industry standard designs obtained from [21].

The paper is organized as follows. Section II covers preliminary concepts related to this paper. Section III describes characteristics of unate functions and their relationship to SAT solutions. In Section IV, we present a recursive, backtracking, algorithm for SAT that exploits the properties of unate functions within the framework of the unate recursive paradigm. In Section V, we present an iterative, non-backtracking search algorithm to solve the SAT problem. The achievements and limitations of these techniques are discussed and experimental results are analyzed. Section VI concludes the paper.

II. Preliminaries

A **binary variable** is a symbol representing a single coordinate of the Boolean space. A **literal** is a variable or its negation (*e.g.* a or $\overline{a}$). A **cube** is a product of literals and it denotes a point, or a set of points, in the Boolean space. A Boolean function is a mapping between Boolean spaces $f : B^n \rightarrow B$. Let $f(x_1, \ldots, x_n)$ be a Boolean function of n variables. The set $\{x_1, x_2, \ldots, x_n\}$ is called the **support** of the function f. The **cofactor** of $f(x_1, \ldots, x_i, \ldots, x_n)$ with respect to variable x_i (the positive cofactor) is $f_{x_i} = f(x_1, \ldots, 1, \ldots, x_n)$. The cofactor of f with respect to variable $\overline{x_i}$ (the negative cofactor) is $f_{\overline{x}_i} = f(x_1, \ldots, 0, \ldots, x_n)$.

The Boole's expansion of a function over a variable (also called Shannon's expansion) is given as follows [22] [23]: Let $f : B^n \rightarrow B$. Then,

$$f(x_1, x_2, \ldots, x_i, \ldots, x_n) = x_i \cdot f_{x_i} + \overline{x}_i \cdot f_{\overline{x}_i} \tag{1}$$

Let f and g be two functions with support variables $\{x_i, i = 1, 2, \ldots, n\}$. Let $\odot$ be an arbitrary binary operator, representing a Boolean function of two arguments. The orthonormal expansion of $f \odot g$ with respect to x_i is given as [24] [23]:

$$f \odot g = x_i(f_{x_i} \odot g_{x_i}) + \overline{x}_i(f_{\overline{x}_i} \odot g_{\overline{x}_i}), \forall i = 1, 2, \ldots, n \tag{2}$$

A function can be represented as a sum of products of n literals, called the **minterms** of the function. Operations on Boolean functions over the same domain can be viewed as operations on the set of their minterms. In particular, sum and product of two functions are the union ($\cup$) and intersection ($\cap$) of their minterm sets, respectively. Implication between two functions corresponds to the containment ($\subseteq$) of their minterm sets.

A function $f(x_1, \ldots, x_i, \ldots, x_n)$ is positive (negative) **unate in variable** x_i if $f_{x_i} \supseteq f_{\overline{x}_i}$ ($f_{\overline{x}_i} \supseteq f_{x_i}$). Otherwise it is **binate** in that variable. A function is unate if it is (positive/negative) unate in all variables in its support. Otherwise it is binate. Let f and g be two functions and x be a variable common to their support. Then, we define f and g to be **consistently unate in variable** x if both functions are either positive unate in x or negative unate in x.

Let f be a function positive unate in variable x. Then, by application of Boole's expansion and the unateness property it can be shown [24] that $f = x \cdot f_x + f_{\overline{x}}$. Similarly, if f is negative unate in x, then $f = f_x + \overline{x} \cdot f_{\overline{x}}$.

From the above, the following observations trivially follow. If a function f is positive unate in x, then if a SAT solution exists for $f = 1$, it can always be found with the assignment $x = 1$. Also, since $f_x \supseteq f_{\overline{x}}$, all SAT solutions in $f_{\overline{x}}$ are *contained* in f_x. Thus, if there does not exist a solution in f_x, there cannot exist a solution in $f_{\overline{x}}$. Hence, a search for a SAT solution in f can be restricted to a search in $x \cdot f_x$. Similarly, if f is negative unate in x, the search for a SAT solution can be restricted to $\overline{x} \cdot f_{\overline{x}}$. These observations form the basis of the satisfiability solutions proposed in the following sections.

III. Unate Functions and Satisfiability

In this section, we demonstrate how we can utilize the properties of unate functions in order to solve the SAT problem efficiently using BDDs. Consider the following problem:

Problem 1: Let there be two functions f and g such that both are *consistently* unate (say, positive unate) in variable x in their support. We are required to solve an instance of the SAT problem given the requirements ($f = 1$) AND ($g = 1$). Furthermore, assume that the BDDs for each f and g can be built within computer memory limitations, while the BDD for their product ($f \cdot g$) cannot be built because of its prohibitively large BDD size. How do we find a satisfying assignment to the input variables without using a backtracking search?

Solution: Let us expand f and g with respect to the variable x, so that $f = x \cdot f_x + f_{\overline{x}}$ and $g = x \cdot g_x + g_{\overline{x}}$. From the orthonormal expansion (2) and the fact that f and g are positive unate in x, we get $f \cdot g = x \cdot f_x \cdot g_x + f_{\overline{x}} \cdot g_{\overline{x}}$. The cofactors of a function are no greater in size (in terms of the number of cubes) than the function itself. Thus, both the products ($f_x \cdot g_x$) and ($f_{\overline{x}} \cdot g_{\overline{x}}$) are no greater in size than the product of f and g. Thus, in order to search for a solution in $f \cdot g$ (a prohibitively large BDD), we can search for solutions in the product of their cofactors ($f_x \cdot g_x$) and ($f_{\overline{x}} \cdot g_{\overline{x}}$), the BDDs for which are smaller in size and can possibly be built. However, it is unnecessary to search for a SAT solution in the product $f_{\overline{x}} \cdot g_{\overline{x}}$, as proved in the following Lemma.

Lemma 1: Let functions f and g be positive unate in variable x. In order to search for a satisfiability solution for $f \cdot g$, it is *sufficient* to search for the solution in the product $x \cdot f_x \cdot g_x$. If f and g are negative unate in x, then it is sufficient to search for a SAT solution in $\overline{x} \cdot f_{\overline{x}} \cdot g_{\overline{x}}$.

Proof: If f and g are positive unate in x, $f_x \supseteq f_{\overline{x}}$ and $g_x \supseteq g_{\overline{x}}$. Thus $f_x \cdot g_x \supseteq f_{\overline{x}} \cdot g_{\overline{x}}$. Hence, if a solution does not exist in $f_x \cdot g_x$, it cannot exist in $f_{\overline{x}} \cdot g_{\overline{x}}$. Also, both f_x and g_x do not contain x in their support. Thus, if $f_x \cdot g_x \neq 0$, then $x \cdot f_x \cdot g_x \neq 0$ and contains the solution. *Mutatis mutandis* for f and g negative unate in x. Fig 2 illustrates the proof by means of a Venn diagram. ■

The above approach can be generalized to an arbitrary number of functions $f^1, f^2, \ldots, f^n$, all consistently unate in variable x. Search for a SAT solution for $(f^1 \cdot f^2 \cdot \ldots \cdot f^n)$ can then be restricted to a search in $x \cdot f^1_x \cdot f^2_x \ldots \cdot f^n_x$ (or $\overline{x} \cdot f^1_{\overline{x}} \cdot \ldots \cdot f^n_{\overline{x}}$, depending on the unateness polarity of x). In case the size of the cofactors is still prohibitively large, we can expand these cofactors with respect to another variable over which all functions are *consistently* unate. Such an iterative cofactoring will eventually lead to functions of relatively smaller size, the product of which can be possibly built. Notice that "backtracks" are eliminated from the search. The technique intelligently prunes the search space by discarding those regions from further exploration for which the search is deemed unnecessary. The above technique can be programmed as a subroutine (SUBROUTINE-USAT) as presented below.

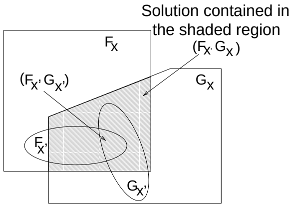

Fig. 2. Containment of SAT solutions in the cofactors: $f_x \cdot g_x \supseteq f_{\overline{x}} \cdot g_{\overline{x}}$.

SUBROUTINE-USAT(f_list)
f_list = list of output function requirements;
solution = 1; /* Initialize the product (solution) */
while TRUE **do**
  get the variable x over which all functions are consistently unate and its unateness polarity;
  /* Polarity is implicit. if +ve polarity, $x = a$. else $x = \overline{a}$. */
  if no more consistently unate variables found **then**
    break;
  end if
  for each function f in f_list **do**
    $f = f_x$;
  end for
  $solution$ = $solution$ AND x;
end while
/* not-so-mammoth product of cofactors */
for each function f in f_list **do**
  $solution$ = f AND $solution$;
end for
SAT assignment = any cube in the BDD representing "$solution$"

Clearly, the above approach is rather limited in scope to find SAT solutions by itself. It is too naive to believe that enough variables can be found over which all functions are consistently unate. In practice, a wide mix of unate and binate functions is found. We need to device a method that can handle a diverse mix of unate and binate functions. In the next section, we present an algorithm that uses the subroutine USAT extensively to solve the SAT problem efficiently.

IV. SAT SOLUTIONS USING THE UNATE RECURSIVE PARADIGM

The recursive approach to handling logic functions has been known for quite some time and its potential demonstrated over a wide variety of applications in logic synthesis [19] [20], such as tautology checking, containment check, complementation, etc. Recursive paradigm applies the orthonormal expansion: $f \cdot g = x \cdot (f_x \cdot g_x) + \overline{x}(f_{\overline{x}} \cdot g_{\overline{x}})$, where f and g are two Boolean functions and x is a variable in their support. The meaning of this expansion is that operations on f and g can be done on a variable-by-variable basis, merging the results of the operation applied to their cofactors with respect to a chosen variable. While many logic functions are not unate, the recursive expansion may lead to cofactors that are unate, and whose processing is very efficient (as demonstrated in section III). In this section, we present a recursive backtracking algorithm to solve the satisfiability problem using the recursive paradigm.

Problem 2: Suppose we need to solve an instance of the satisfiability problem, given certain value requirements for n functions $f^1, f^2, \ldots, f^n$. Assume that their product cannot be built because of BDD size limitations. Assume, further, that these functions are highly binate so that we cannot efficiently apply the subroutine USAT presented in the previous section. How do we solve the SAT problem?

Solution: In order to generate smaller functions so as to be able to build their product, let us expand all these functions w.r.t. some variable. The choice of splitting variable is important. Brayton *et al.* in [19] proposed a heuristic that consists of selecting binate variables that are most likely to create unate sub-problems. We follow a similar approach. For each input variable x, we evaluate how many output functions are binate in x. We select that variable over which most functions are binate. We define this variable as the **most active binate** variable. After expanding all the functions $f^1, \ldots, f^n$ over x, we search for the solution in the positive cofactors. If the solution is not found, we backtrack and search for the solution in the negative cofactors. The above process is applied recursively until no more binate variables are found or the (user defined) recursion depth limit has been reached. At the end of the recursion tree, if no more binate variables are found, it means that all resulting functions (cofactors) are unate, and we can now use subroutine USAT to efficiently search for the SAT solution. If not all resulting cofactors are unate, the subroutine attempts to build the product of all functions. This product can possibly be built, now that the BDDs representing these cofactors are significantly smaller than the original functions. This approach was programmed as an algorithm called BSAT (splitting over Binate variables) which is described in Algorithm 1. Using this approach, we were able to obtain SAT solutions for all benchmark examples (shown in Table II), including those for the 16×16 multiplier.

```
f_list = f^1, f^2, ..., f^n;
BSAT( f_list )
        x = next most actively binate variable;
        if no more binate variables OR max recur depth then
              /* terminal case */
              result = SUBROUTINE-USAT( f_list );
              return result;
        endif
        cof_f_list = f^1_x, f^2_x, ..., f^n_x;
        result = BSAT( cof_f_list );
        result = result AND x ;
        if solution found then return result;
        cof_f_list = f^1_x̄, f^2_x̄, ..., f^n_x̄;
        result = BSAT( cof_f_list );
        result = result AND x̄ ;
        return result;
```

Algorithm 1: BSAT: Satisfiability using the unate recursive paradigm.

A. Analyzing the Results

The experiments were carried out over a wide range of benchmark circuits and are reported in Table II under the BSAT column. The performance of BSAT is compared with that of GRASP[8] and the basic BDD-SAT approach [14] [13]. The performance of SAT tools for instances for which solutions exist varies markedly as compared to that for which a satisfying assignment is infeasible. For this reason, the data for feasible and infeasible instances is reported in separate columns (as "Sol." and "No Sol.", respectively). Furthermore, in order to cover a wide range of solutions, the CPU times reported for feasible solutions are averaged over 20 different instances of the SAT problem (except for the last 3 designs). The version of GRASP [8] available to us only checks whether or not a problem is satisfiable. It does not provide input assignments for feasible SAT solutions. So, the CPU times reported for GRASP correspond to the time required to check whether or not a SAT solution exists. The column "Max. BDD nodes" corresponds to the maximum BDD-size (in terms of BDD nodes) observed over all intermediate products for all problem instances. Due to strict space limitations for this paper, the results for smaller circuits, the product of whose output requirements could be easily built, are not shown. We were unable to experiment with GRASP on the benchmarks received from [21] because of data/netlist incompatibility problems. For the same reason we have not been able to report results for the DIMACS benchmarks.

It is apparent from the results that the maximum BDD size of the intermediate computations in BSAT is far smaller than that in the

TABLE II

COMPARISON OF PERFORMANCE OF VARIOUS SAT ENGINES. CPU TIMES AVERAGED OVER 20 DIFFERENT INSTANCES OF SAT PROBLEMS.

			GRASP		BDD-SAT: BDD Product				BSAT				IS-USAT			
			CPU (s)		CPU (s)		Max BDD size		CPU (s)		Max BDD size		CPU (s)		Max BDD size	
Benchmark	#PI	#PO	Sol	No sol	Sol	No Sol	Sol	No Sol	Sol	No Sol	Sol	No Sol	Sol	No Sol	Sol	No Sol
C1355	41	35	2	9	6	4	6,786	3,718	233	317	4,851	559	6.5	9	6,740	1,047
C7552	207	108	3	17	40	6.5	415K	16K	24	44	40K	33K	31	4.5	214K	139K
C5315	178	123	2	2	435	133	1,920K	840K	62	840	60K	44K	31	18	21K	18K
11×11mult	22	22	185	44	12	6.5	35K	2K	15	133	379	17K	8.9	4	34K	25K
13×13mul	26	26	813	319	102	16	423K	52K	137	520	2K	40K	109	49	401K	47K
32-dpath	73	32	483	–	106	–	315K	–	15	–	67K	–	12.2	–	18K	–
pair	173	137	1.8	1	136	3	382K	23K	410	761	112K	14K	46	5	92K	1K
des	256	245	abort	abort	–	–	no mem	no mem	967	1839	18K	69K	109	48	624K	148K
16×16mul	32	32	2920	417	–	–	no mem	no mem	6216	11320	331K	761K	–	–	no mem	no mem
Cubic[8]	28	2	–	–	1	–	1.2K	–	4	–	533	–	2	–	429	–
Square[10]	30	2	–	–	3	–	1.3K	–	3	–	392	–	4	–	670	-
Square[16]	48	2	–	–	–	–	no mem	no mem	1102	–	894K	–	183	–	1,478K	–

basic BDD-SAT approach. For this reason, BSAT is able to compute SAT solutions for large designs where BDD-SAT fails. However, for instances of SAT for which solutions do not exist, the performance of BSAT is far from satisfactory. The excessive computation times required by BSAT can be attributed to the (semi-)exhaustive nature of the search process. In majority of the infeasible instances of SAT, the basic BDD-SAT outperforms BSAT when SAT instances do not have solutions.

SAT solutions for the 16×16 multiplier need further explanation. It is well known that BDDs for the 16×16 (and higher) multipliers cannot be built in terms of primary input variables. We use partitioned ROBDDs [18] with intermediate variables to represent the Boolean functions for the primary outputs. Only the primary input variables are used to carry out the recursive expansion. At the bottom of the recursion tree, we substitute the intermediate variables in these cofactors to obtain BDDs in terms of the primary inputs and then invoke SUBROUTINE-USAT. If the size of the resulting cofactors is still relatively large, the substitution of intermediate variables can be prohibitive in both space and time. In order to simplify the problem, we perform a disjunctive decomposition of these cofactors into smaller BDDs ($f = f_1 + f_2$) according to [25], and then substitute the intermediate variables. Since these decomposed BDDs are much smaller, the substitution process is quite fast, as verified by experimentation. The authors wish to point out that the algorithm BSAT is not particularly targeted for solving SAT problems for multipliers. There exist more efficient techniques [12] [26] that are geared specifically toward solving the SAT problem for multipliers. We just wish to demonstrate that, to a certain extent, this algorithm does not suffer from BDD size explosion problems. While BSAT is able to successfully compute SAT solutions for all benchmarks, it spends a lot of time backtracking for solutions that are either hard to find or do not exist at all. In the following section, we present an algorithm that attempts to overcome this deficiency.

V. AN INCOMPLETE SEARCH FOR SAT SOLUTIONS

It is desired of satisfiability tools that if solutions exist, they should be found quickly. Also, if a problem is undecidable, SAT tools should be quick to evaluate infeasibility. Recent works [8] [9] employ intelligent conflict analysis procedures and learning techniques [27] that enable early detection of undecidability/infeasibility, thus enhancing the overall performance of SAT tools.

Analysis of the results presented in Table II reveals that BDD-SAT is effective in detecting infeasibility of SAT instances. If a solution does not exist, the product of all the constraints equals zero, and BDD-SAT terminates quickly for almost all benchmarks. BSAT, being an exhaustive search, spends a lot of time and effort backtracking throughout the search space to prove infeasibility. On the other hand, when SAT solutions do exist, BDD-SAT suffers from BDD size explosion problems for large designs, whereas BSAT arrests the growth of BDD size and is able to compute the SAT solution. It would be desirable to develop a technique that has the advantages of both BDD-SAT and BSAT, while overcoming their respective limitations. We now present a search technique, called INCOMPLETE-SEARCH-USAT (abbreviated IS-USAT), that attempts to leverage the advantages of the above techniques.

Instead of searching for SAT solutions over the complete search space, IS-USAT explores only those regions that have a *high likelihood* of containing a solution, discarding the rest. Hence, an incomplete search. Storing a restricted search space in BDDs avoids BDD-size explosion problems and further reduces the time to search for a solution. However, for some SAT instances a feasible solution may exist only in the regions discarded from exploration. IS-USAT would then erroneously classify this instance as infeasible. This is a penalty that we may have to pay by exploring only a part of the search space. For certain applications, such as ATPG and automatic functional vector generation, sacrificing a few feasible solutions in lieu of fast computation of easy solutions may be an acceptable behaviour. However, this emphasizes the need for an intelligent technique to guide the search so that search space containing significant number of solutions is not discarded. In what follows, we present an intelligent heuristic that identifies regions to search that have a high likelihood of containing a solution. The heuristic exploits properties of unate functions (namely, unate containment) to guide the search. We describe the intuition behind this approach and demonstrate its effectiveness via experiments over a wide range of benchmarks.

Problem 3: Let f and g be two functions such that f is unate (say, positive unate) in x and g is binate in x. Assume that the product of f and g cannot be built because the BDD size of the product exceeds the memory limits. How do we find a satisfying assignment to the inputs without using a backtracking search?

Solution: Let us just expand the unate function f over variable x and leave g intact. Then $f = x \cdot f_x + f_{\overline{x}}$. The product $f \cdot g = (x \cdot f_x + f_{\overline{x}}) \cdot g = x \cdot f_x \cdot g + f_{\overline{x}} \cdot g$. As $f_x \supseteq f_{\overline{x}}$, $f_x \cdot g \supseteq f_{\overline{x}} \cdot g$, the cubes (SAT solutions) contained in $f_{\overline{x}} \cdot g$ are also contained in $f_x \cdot g$. This may suggest that the regions corresponding to $f_{\overline{x}} \cdot g$ could be discarded. Clearly, if $f_x \cdot g = 0$, then because $f_x \cdot g \supseteq f_{\overline{x}} \cdot g$, $f_{\overline{x}} \cdot g$ is also equal to 0 (due to unateness of f) and the solution does not exist. This underlines the intuition behind discarding the region $f_{\overline{x}} \cdot g$ from further exploration. However, the binateness of g can create problems. Because the literal $\overline{x}$ may be present in g, the product $x \cdot g$ may be equal to 0. In such a case, we have to search for the SAT solution in $f_{\overline{x}} \cdot g$ and discard $x \cdot f_x \cdot g$. If $x \cdot g \neq 0$, then we search for the solution in $x \cdot f_x \cdot g$. The decision to guide the search can now be made w.r.t. the product $x \cdot g$.

Now consider the following case: $x \cdot g \neq 0$, $f_x \cdot g \neq 0$, but $x \cdot f_x \cdot g = 0$. The solution may be contained in $f_{\overline{x}} \cdot g$, but according to our decision process, this region is discarded. Our technique would then incorrectly suggest infeasibility. However, experimental evidence suggests that such cases are relatively infrequent.

The above process can be generalized to an arbitrary number of functions $f^1 \cdot \ldots \cdot f^n$. First, we identify all those functions that are consistently unate in variable x (*i.e.* all those functions that are either positive unate or negative unate in x). The remaining functions are separated and their product g is built. If all functions are consistently unate in x, Lemma 1 applies. Otherwise, a decision is made according to the product $x \cdot g$ (or $\overline{x} \cdot g$ depending upon the unateness polarity of x) to guide the search for a solution. These concepts form the basis of the algorithm (IS-USAT) presented in Algorithm 2. Central to the algorithm is a heuristic technique to identify the variable x, over which the unate functions are to be expanded. For each input variable x, we evaluate how many output functions are positive and negative unate in x. The variable over which most functions are *consistently* unate, is selected as the candidate over which the expansion is carried out. We define this variable as the **most active unate** variable. For all those functions that are not unate in variable x w.r.t. its unateness polarity, we build their product. Depending upon the polarity of the most active unate variable x, the expansion $f = x \cdot f_x + f_{\overline{x}}$ or $f = f_x + \overline{x} \cdot f_{\overline{x}}$ is performed for all consistently unate functions. A decision is made according to the outcome of the product $x \cdot g$ (or $\overline{x} \cdot g$), and the search is guided. The above process is carried out in each iteration, until no more active unate variables are found, or the (user defined) iteration limit has been reached.

Experimental data corresponding to IS-USAT is presented in Table II. As before, the CPU times for feasible SAT solutions are averaged over 20 different instances. Other than the 16×16 multiplier, we were able to find SAT solutions for all benchmarks. The results depict that "Max. BDD nodes" observed over all intermediate computations is well within manageable limits. IS-USAT rates better in CPU time performance than both BDD-SAT and BSAT. Most important of all, IS-USAT incorrectly reported infeasible solutions for only **three** (actually feasible) SAT instances, one each corresponding to benchmarks *des*, *pair* and *C5315*. In all other SAT instances, feasible solutions were found quickly. While this may not reflect the accuracy of the search heuristics in IS-USAT, it is quite encouraging to observe that the loss of information by discarding regions of search space is not significant.

IS-USAT would not be very effective if most functions were binate in the variables in their support. When consistently unate variables are not found, INCOMPLETE-SEARCH-USAT attempts to build the conjunction of all the constraints, thus reducing to BDD-SAT. In fact, for multipliers, almost all functions are binate. For this reason, maximum size of the intermediate product BDDs observed for multipliers using IS-USAT is almost the same as that observed with BDD-SAT. It is also for this reason that we were unable to solve SAT for 16×16 multiplier.

VI. Conclusions and Future Work

This paper has presented two efficient algorithms to solve the Boolean satisfiability problem using ROBDDs. The first algorithm demonstrates the application of the well known *unate recursive paradigm* to Boolean satisfiability. Using our approach, we were able to obtain SAT solutions not only for all benchmarks in the ISCAS'85 and MCNC benchmark suite, but also for other large practical designs. We are in the process of incorporating learning techniques in our recursive algorithm BSAT. Without learning techniques,

```
f_list = list of output function requirements;
g = 1; /* Initialize the binate product */
while TRUE do
  get the most actively unate variable x and its unateness polarity;
  /* Polarity is implicit. if +ve polarity, x = a. else x = ā. */
  if no more unate variables found then
    break;
  end if
  for each function f in f_list do
    /* separate functions unate and non-unate in x */
    if f is not unate in x w.r.t. unateness polarity then
      g = f AND g; /* build the binate product */
      remove f from f_list;
    end if
  end for
  /* now f_list only has functions consistently unate in x */
  if x AND g == 0 then
    for each function f in f_list do
      f = f_x̄;
    end for
  else
    for each function f in f_list do
      f = f_x;
    end for
    g = g AND x;
  end if
end while
/* not-so-mammoth product of cofactors & binate functions */
for each function f in f_list do
  g = f AND g;
end for
SAT assignment = any cube in g
```

Algorithm 2: INCOMPLETE-SEARCH-USAT: Incomp. SAT search.

BSAT spends a lot of time backtracking for solutions that are either hard to find or do not exist. We believe that learning techniques incorporated with a non-chronological backtracking strategy would speed up the search process in BSAT manifold.

The second algorithm, INCOMPLETE-SEARCH-USAT, presented in the paper also exploits characteristics of unate functions to search for SAT solutions. The search is *incomplete* inasmuch as only those regions of the search space are explored that have a high likelihood of containing a solution. While this technique cannot be relied upon to prove unsatisfiability in unequivocal terms, it is able to compute SAT solutions quickly for a large percentage of SAT instances. The experimental results clearly demonstrate the memory efficiency of the proposed algorithms.

REFERENCES

[1] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

[2] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory", *Journal of the ACM*, vol. 7, pp. 201–215, 1960.

[3] C. E. Blair and *et al.*, "Some Results and Experiments in Programming Techniques for Propositional Logic", *Comp. and Oper. Res.*, vol. 13, pp. 633–645, 1986.

[4] J. W. Freeman, "Improvements to Propositional Satisfiability Search Algorithms", *Ph.D. Dissertation, Dept. of Comp. and Inf. Sc., Univ. of Penn.*, vol. , May 1995.

[5] T. Larabee, *Efficient Generation of Test Patterns using Satisfiability*, PhD thesis, Dept. of Computer Science, Stanford University, Feb. 1990.

[6] P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Combinational Test Generation using Satisfiability", Technical Report UCB/ERL M92/112, Dept. of EECS., Univ. of California at Berkeley, Oct. 1992.

[7] R. Zabih and D. A. McAllester, "A Rearrangement Search Strategy for Determining Propositional Satisfiability", *in Proc. Natl. Conf. on AI*, 1988.

[8] J. Marques-Silva and K. A. Sakallah, "GRASP - A New Search Algorithm for Satisfiability", *in ICCAD'96*, pp. 220–227, Nov. 1996.

[9] L. G. Silva, L. M. Silvera, and J. Marques-Silva, "Algorithms for Solving Boolean Satisfiability in Combinational Circuits", *in Proc. DATE*, pp. 526–530, March 1999.

[10] R. E. Bryant, “Graph Based Algorithm for Boolean Function Manipulation”, *IEEE Transactions on Computers*, vol. C-35, pp. 677–691, 1986.

[11] K. S. Brace, R. Rudell, and R. E. Bryant, “Efficient Implementation of the BDD Package”, *Proceedings of the Design Automation Conference*, pp. 40–45, 1990.

[12] P. Ashar and *et al.*, “Boolean Satisfiability and Equivalence Checking using General Binary Decision Diagrams”, *in Proc. ICCD*, 1991.

[13] S. Jeong and F. Somenzi, “A New Algorithm for the Binate Covering Problem and its Application to the Minimization of Boolean Relations”, *in ICCAD*, 92.

[14] B. Lin and F. Somenzi, “Minimization of Symbolic Relations”, *in ICCAD*, 90.

[15] T. Villa and *et al.*, “Explicit and Implicit Algorithms for Binate Covering Problems”, *IEEE Trans. CAD*, vol. Vol. 16, pp. 677–691, July 1997.

[16] A. J. Hu, G. York, and D. L. Dill, “New Techniques for Efficient Verification with Implicitly Conjoined BDDs”, *in Proc. DAC*, pp. 276–282, 1994.

[17] A. J. Hu and D. L. Dill, “Reducing BDD Size by Exploiting Functional Dependencies”, *in Proc. DAC*, pp. 266–271, 93.

[18] A. Narayan and *et al.*, “Partitioned ROBDDs: A Compact Canonical and Efficient Representation for Boolean Functions”, *in Proc. ICCAD*, ’96.

[19] R.K. Brayton, C. McMullen, G.D. Hachtel, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.

[20] R. Rudell and A. Sangiovanni-Vincentelli, “Multiple-valued Minimization for PLA Optimization”, *IEEE Tr. on CAD*, vol. CAD-6, pp. 727–750, Sept. 1987.

[21] C. L. Huang, “Private Communication”, Avery Design Systems, Inc.

[22] F. Brown, *Boolean Reasoning*, Kluwer Academic Publishers, 1990.

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

[24] G. DeMicheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 94.

[25] G. Cabodi and *et al.*, “Disjunctive Partitioning and Partial Iterative Squaring: An Effective Approach for Symbolic Traversal of Large Circuits”, *in Proc. DAC*, ’97.

[26] F. Fallah, S. Devadas, and K. Keutzer, “Functional Vector Generation for HDL models using Linear Programming and 3-Satisfiability”, *in Proc. DAC*, ’97.

[27] W. Kunz and D.K. Pradhan, “Recursive Learning: A New Implication Technique for Efficient Solutions to CAD Problems – Test, Verification and Optimization”, *IEEE Tr. on CAD*, vol. 13, pp. 1143–1158, Sep. 1994.