

# Satometer: How Much Have We Searched?

Fadi A. Aloul, *Student Member, IEEE*, Brian D. Sierawski, and Karem A. Sakallah, *Fellow, IEEE*

**Abstract**—We introduce Satometer, a tool that can be used to estimate the percentage of the search space actually explored by a backtrack Boolean satisfiability (SAT) solver. Satometer calculates a normalized minterm count for those portions of the search space identified by conflicts. The computation is carried out using a zero-suppressed binary decision diagram data structure and can have adjustable accuracy. The data provided by Satometer can help diagnose the performance of SAT solvers and can shed light on the nature of a SAT instance.

**Index Terms**—Algorithms, Boolean algebra, Boolean functions, computer-aided design (CAD), design automation, formal logic, logic, logic functions, search methods.

## I. INTRODUCTION

THE LAST few years have seen significant algorithmic advances in, and carefully crafted implementations of, Boolean satisfiability (SAT) solvers [2], [15], [21], [24], [27], [29]. This has led to their successful application to a wide range of large-scale electronic design automation (EDA) problem instances consisting of thousands of variables and millions of clauses [3], [12], [16], [22], [25], [26]. Despite these remarkable developments, SAT solvers cannot escape the underlying worst-case exponential complexity of their search space and must sometimes be aborted after a certain time-out limit has been reached. Typically, when a solver aborts it provides very little data about how much progress it had achieved up to that point. Such data can be quite useful. Knowing, for instance, that the solver had managed, after several hours, to explore only 1% of the search space might suggest a very hard problem instance and the need, perhaps, to try a different approach. If, on the other hand, the solver reports exploring more than 99% of the search space without finding a solution, it may be reasonable to assume that the instance has very few satisfying assignments or is possibly unsatisfiable.

Satometer (pronounced like barometer) is an accessory that can be used with any backtrack search SAT solver to report the percentage of search space actually explored by the solver. It requires the solver to emit the set of clauses corresponding to the conflicts encountered during the search. It can be used dynamically, while the SAT solver is running, to indicate progress in the search for a solution. It is more useful, however, as a postprocessor to analyze the result of an aborted or completed search.

Manuscript received June 24, 2002; revised November 27, 2002 and February 5, 2003. This paper was recommended by Guest Editor L. Lavagno. This work was supported in part by the DARPA/MARCO Gigascale Silicon Research Center, in part by an Agere Systems/SRC Research fellowship, and in part by the National Science Foundation under Grant 0205288. This paper was recommended by Guest Editor L. Lavagno.

The authors are with the Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, MI 48109-2122 USA (e-mail: faloul@umich.edu; bsieraws@umich.edu; karem@umich.edu).

Digital Object Identifier 10.1109/TCAD.2003.814960

The paper is organized as follows. In Section II, we present an overview of SAT. This is followed by a summary of previous work in Section III. In Section IV, we introduce our measure of search progress. We then describe, in Section V, how this measure can be computed using binary decision diagrams (BDDs) and zero-suppressed BDDs (ZBDDs). In Section VI, we illustrate the utility of this measure in a variety of experimental scenarios and evaluate the performance of Satometer. We conclude in Section VII with a summary of the paper's main contributions.

## II. PRELIMINARIES

A Boolean formula  $\varphi$  given in *conjunctive normal form* (CNF) consists of a conjunction of clauses. Each clause is a disjunction of literals, where a literal is either a variable  $x$  or its negation  $\neg x$ . A clause is *satisfied* if at least one of its literals is set to 1, *unsatisfied* if all its literals are set to 0, *unit*, if all but one literal are set to 0, and *unresolved*, otherwise. Consequently, a formula is satisfied if all its clauses are satisfied, and unsatisfied if at least one clause is unsatisfied. The goal is to identify a set of assignments for variables that would satisfy the formula or prove that no such assignment exists and that the formula is unsatisfiable.

Backtrack search solvers have been shown to be very robust in solving hard, real-world SAT instances [15], [21]. The Davis, Logemann, and Loveland (DLL) search procedure [8] provides the basis for the majority of backtrack search algorithms. The procedure performs a depth-first exploration of the search space by recursively: 1) making a *decision assignment*, i.e., selecting, according to some branching heuristic, an unassigned variable and setting its value to either 1 or 0; 2) deriving additional assignments, referred to as *implications*, to other variables due to the decision assignment(s); and 3) systematically backtracking from *conflicts*, i.e., assignment that cause the formula to become unsatisfied, in order to make other (untried) decisions. Implications are triggered by the *unit-clause rule* which sets the only unassigned literal in a unit clause (see the previous definition) to the only value that would satisfy the clause, namely 1. Repeated application of this rule is referred to as Boolean constraint propagation (BCP) [21]. The procedure terminates and proves satisfiability when all clauses are satisfied or unsatisfiability when no new variable assignment can be made without producing an unsatisfied clause.

Recently, several enhancements to the DLL approach have been proposed. Among the various enhancements, *conflict analysis*, which was introduced in GRASP [21], was shown to significantly prune the search space. The procedure is called after each conflict to analyze its causes and to generate adequate information—in the form of additional so-called *conflict-induced clauses*—to prevent the conflict from recurring in other parts of the search space.

### III. PREVIOUS WORK

Researchers have always been interested in measuring the search progress of backtrack search solvers. Ideally, the best measure of search progress is an accurate estimate of the amount of time needed to complete the search. Despite the predictive nature of this problem and the difficulty of computing an exact answer, several attempts have been made to measure search progress.

The first attempt to predict the time needed by a backtrack search program to complete was proposed by Knuth [10]. In this approach, iterative sampling is used to estimate the size of the decision tree that will be searched to solve the problem. The basic idea is to explore a single random path from the root to a leaf node. Assuming that all nodes, at decision level  $i$ , have the same number of successors, denoted as  $d_i$ , the number of nodes in the decision tree can be estimated using the formula

$$1 + d_1 + d_1d_2 + d_1d_2d_3 + \dots \quad (1)$$

The estimate is improved by averaging over a number of iterations.

More recent work by Purdom [17] extended Knuth's algorithm with *partial backtracking*. The idea is to traverse  $m$  successor nodes for each visited node as opposed to traversing a single successor node as in Knuth's algorithm. Purdom showed that this modification is more effective on deep trees. The proposed algorithm is identical to Knuth's algorithm when  $m = 1$  and follows a complete backtracking approach when all successors are traversed. The tradeoff between the estimation accuracy and the runtime overhead depends on the value of  $m$ .

Knuth's algorithm was also extended by Chen [6]. Chen's approach is based on *heuristic sampling* of the nodes in the decision tree, in which a cost function is associated with each node. The goal is to estimate the cost of processing the complete set of nodes in the decision tree.

In the SAT context, all three tree size estimation attempts are based on the assumption of a *static* clause database, i.e., the clause database is assumed to be fixed during the search process. It is therefore unclear how such estimation methods can be extended to handle the recent backtrack search programs that *dynamically* augment the clause database using clause-learning techniques, such as conflict analysis [21].

A recent effort that handles a dynamic clause database was proposed by Kokotov and Shlyakhter [11]. In their approach, a progress bar is integrated into a backtrack SAT solver to estimate the time left to complete the search. The bar is updated based on either *historical* or *predictive* estimates of the size of the decision tree maintained by the SAT solver. Historical estimators are based on averaging the time spent on nodes at the same decision level of the node being examined. The assumption is that particular, structured problems, are likely to share similar subtree sizes between nodes at the same decision level. They propose to further improve the average by weighting it according to the distance between the nodes. The predictive estimators ignore the subtree sizes of previous nodes and instead focus on estimating the needed runtime by analyzing the unresolved clauses. The premise is that variables that eliminate more

clauses, either by satisfying clauses directly or implying other assignments, are likely to require a smaller subtree size which needs less time to explore. They reported that the bar is able to predict progress with an accuracy of 80%–90% without significantly impacting the solver's run time. Their approach, however, provides no guarantees to confine the accuracy of the results.

The metric we propose in this paper is different than all previous metrics in that it tries to capture the size of the search space that has already been explored. Thus, it is a *retrospective* rather than a *prospective* measure of search progress and does not predict how much more time is required to finish the search. It is important to note that these two metrics are complementary, in the sense that measuring the size of the covered search space is viewed as an additional piece of information that along with an estimate of the time required to complete the search can provide a more complete picture of the performance of the solver. In particular, such a retrospective metric can be used to analyze the result of an aborted search. We noticed that a SAT solver that manages to explore only a small portion of the search space after running for a long amount of time, *might* suggest a hard problem instance. For example, the state-of-the-art SAT solver Chaff [15] was unable to solve the *k2\_fix\_gr\_rcs\_w9* instance [16] (shown in Fig. 6) after running for 500 s. It explored less than 45% of the search space which might indicate that the instance is hard to solve. In fact, Chaff was still unable to solve the problem after running for 10 h. On the other hand, a solver that explores a high percentage of the search space without finding a solution, might suggest an instance with few satisfying assignments or an unsatisfiable instance. Fig. 6 shows an example in which Chaff was able to explore almost 99.7% of the search space for the *9\_vliw\_bp\_mc* instance [26]. Upon completing the search, the instance was proven to be unsatisfiable. In the following sections, we will describe the space coverage metric and show how it can be efficiently computed.

### IV. SEARCH SPACE COVERAGE

In our approach, we view the search process as a sequence of moves that continually (and systematically) modify a (partial) variable assignment until: 1) a satisfying assignment (a solution) is found; 2) the formula is proven to be unsatisfiable (has no solution); or 3) a time-out limit is reached. Along the way, many assignments that are explored will correspond to zeros of the function represented by the formula and will cause the search process to backtrack. Every time such a "conflict" occurs, it identifies a portion of the search space that can be regarded as *explored* and found to contain no solutions.

Let  $A_1, A_2, \dots, A_i$  denote the assignments that correspond to the first  $i$  conflicts. We can measure how much of the search space has been explored by counting the number of minterms<sup>1</sup> covered by the function  $A_1 + A_2 + \dots + A_i$ . Normalizing this count by the total size of the space yields the percentage of the space that has been explored up to this point. We will use the notation  $\|f\|$  to express the normalized number of minterms of the function  $f$ . Thus,  $\|a + b\| = 75\%$ ,  $\|a \cdot b\| = 25\%$ , and  $\|a \oplus b\| = 50\%$ . In the sequel, we will refer to  $\|f\|$  as the *size* of  $f$ .

<sup>1</sup>A minterm is a complete truth assignment that sets the function to 1.

Decisions	Impli- cations	Conflicts		Explored Space	
		Y/N	Clause	Minterms	%
1	$a$	N			
2	$ab$	N			
3	$abc$	$d'$	$(a' + b' + c')$	2	12.5
4	$abc'$	$d$	$(a' + b' + c)$	4	25
5	$ab'$	$c'd$	$(a' + b)$	8	50
6	$a'$	N			
7	$a'b$	N			
8	$a'bc$	$d'$	$(a + b' + c')$	10	62.5
9	$a'bc'$	$d$	Solution!		

(a) Execution trace of a basic backtrack SAT Solver

Decisions	Impli- cations	Conflicts		Explored Space	
		Y/N	Clause	Minterms	%
1	$a$	N			
2	$ab$	N			
3	$abc$	$d'$	$(b' + c')$	4	25
4	$ab$	$c'd$	$(a' + b')$	6	37.5
5	$a$	$b'c'd$	$(a')$	10	62.5
		$a'$	N		
6	$b$	$a'c'd$	N	Solution!	

(b) Execution trace of a conflict-based backtrack SAT Solver

 Fig. 1. Execution traces of two different SAT solvers on the formula in (2) illustrating how search space coverage is measured. In the decision column,  $abc'$  means that 1 was assigned to  $a$  and  $b$ , and 0 was assigned to  $c$ .

This measure can be equivalently computed by considering the *conflict-induced clauses* identified at each conflict. Let  $C_1, C_2, \dots, C_j$  denote the conflict-induced clauses identified after the first  $i$  conflicts. In general,  $j \geq i$  as one or more conflict-induced clauses may be identified at each conflict. The portion of the search space that would have been explored after processing the  $i$ th conflict can now be computed as  $1 - \|C_1 \cdot C_2 \cdot \dots \cdot C_j\|$ .

An illustration of these computations is shown in Fig. 1 for the four-variable formula

$$\varphi = (a + b + c) \cdot (a + b + c') \cdot (a' + b + c') \cdot (a + c + d) \cdot (a' + c + d) \cdot (a' + c + d') \cdot (b' + c' + d') \cdot (b' + c' + d). \quad (2)$$

## V. COMPUTATION OF SEARCH SPACE COVERAGE

When the conflicting assignments are disjoint, i.e., when  $A_k \cdot A_l = 0$  for  $k \neq l$ , space coverage can be simply calculated by the formula

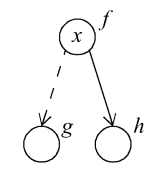
$$\|A_1 + A_2 + \dots + A_i\| = \sum_{1 \leq k \leq i} \|A_k\|. \quad (3)$$

Equivalently, if the conflict-induced clauses are disjoint, i.e., if  $C_k + C_l = 1$  for  $k \neq l$ , then space coverage is simply

$$1 - \|C_1 \cdot C_2 \cdot \dots \cdot C_j\| = \sum_{1 \leq k \leq j} (1 - \|C_k\|). \quad (4)$$

In other words, if conflicts identify nonoverlapping portions of the search space, then the size of the explored space can be found by simply adding the sizes of the different portions. In general, this will only apply to standard backtrack algorithms that do not employ conflict diagnosis to prune the search space. To compute the size of the explored space when conflict diagnosis is employed, we have no choice but to build some type of symbolic representation for the disjunction of conflict assignments or the conjunction of conflict-induced clauses. We describe below the

 TABLE I  
SEMANTICS OF DECISION DIAGRAMS

Diagram Type	Internal Nodes		Terminal Nodes	
	Representation		$0^f$	$1^f$
BDD			$f = 0$	$f = 1$
ZBDD	Set	$f = g \cup (\{x\} \times h)$	$f = \emptyset$	$f = \{\emptyset\}$
	CNF	$f = (g) \cdot (x + h)$	$f = 1$	$f = 0$
	DNF	$f = (g) + (x \cdot h)$	$f = 0$	$f = 1$

two representations we examined and show how we used them to measure space coverage. Without loss of generality, we restrict the discussion to building representations for conjunctions of conflict-induced clauses.

### A. Coverage Computation Using BDDs

The conflict-induced clauses can be symbolically “ANDed” using a reduced ordered binary decision diagram (ROBDD or BDD for short) [4]. BDD semantics allow us to write the function  $f$  at a node labeled with variable  $x$  using Boole’s expansion

$$f = x' \cdot g + x \cdot h \quad (5)$$

where  $g$  and  $h$  are the functions associated with the 0- and 1-children of that node (see Table I). This immediately leads to the following formula for the size of  $f$ :

$$\|f\| = \frac{1}{2}(\|g\| + \|h\|). \quad (6)$$

The size of the function represented by a BDD can now be obtained by sweeping the BDD from the terminal nodes toward the top node and applying (6) at each visited node. The sweep is initialized by setting  $\|0\| = 0$  and  $\|1\| = 1$  for the constant functions of the terminal nodes.

### B. Coverage Computation Using ZBDDs

The problem with the BDD representation, of course, is that it quickly runs out of memory. An alternative that has lower memory requirements is the ZBDD originally proposed by Minato [14] for manipulating large combination sets, including sets of Boolean cubes. A combination set  $S$  can be regarded as a *set of sets*, e.g.,  $\{\{a, b\}, \{c, d, e\}, \{a, d\}, \{b\}\}$ . Recently, Chatalic and Simon [5] demonstrated that ZBDDs can be an effective implicit representation of large CNF formulas and showed how they can be used to perform “multiresolution” to solve some large structured SAT instances. In this scenario, the aforementioned example set corresponds to the CNF formula  $(a + b) \cdot (c + d + e) \cdot (a + d) \cdot (b)$ , i.e., each combination is viewed as an OR term (a clause) and the entire set (a union of combinations) as an AND term. Such an interpretation allows the semantics of Boolean algebra to be layered on top of the semantics of set algebra to obtain further compression of the ZBDD structure. In particular, Chatalic and Simon extended the standard ZBDD set-union operation to a subsumption-free union that automatically removes any clause that is completely subsumed by another clause. In the above example, combination  $\{a, b\}$  is subsumed by combination  $\{b\}$  yielding the *logically equivalent* set  $\{\{c, d, e\}, \{a, d\}, \{b\}\}$ . Additional reduction rules based on literal absorption, i.e.,  $(a) \cdot (a' + b + c) = (a) \cdot (b + c)$ , were subsequently described in [1].

The semantics of ZBDD nodes were first articulated by Lobbing *et al.* in [13]. Given a set of atoms  $\{a, b, c, \dots\}$ , a ZBDD node labeled with atom  $x$  represents a combination set  $f$  constructed according to the formula

$$f = g \cup (\{x\} \times h) \quad (7)$$

where  $g$  and  $h$  are the combination sets associated with the 0- and 1-children of that node (see Table I). The terminal 0 and 1 nodes correspond, respectively, to the empty set (set of no combinations) and to the set of consisting of the empty combination. The “product” in (7) is similar to the Cartesian product of two sets and is defined by

$$S \times T = \bigcup_{s \in S, t \in T} \{s \cup t\}. \quad (8)$$

For example, given the combination sets  $S = \{\{a, b\}, \{b, c\}\}$  and  $T = \{\{a, d\}, \{e\}\}$ , their product is

$$S \times T = \{\{a, b, d\}, \{a, b, e\}, \{a, b, c, d\}, \{b, c, e\}\}. \quad (9)$$

It is important to note that  $S \times T \neq S \cup T$ . Thus, for this example,  $S \cup T = \{\{a, b\}, \{b, c\}, \{a, d\}, \{e\}\}$ .

When used to represent a CNF formula, the formula  $f$  associated with a ZBDD node labeled by variable  $x$  follows the same template of (7) except that the union of atoms in a combination

is viewed as logical OR and the union of the combinations is viewed as logical AND yielding

$$f = (g) \cdot (x + h) \quad (10)$$

where  $g$  and  $h$  are the formulas associated with the 0- and 1-children of that node (see Table I). The terminal 0 and 1 nodes, correspond, respectively, to the constant 1 and constant 0 functions.

To represent CNF formulas with ZBDDs, the set of atoms is taken to be the set of literals over which the formula is defined. In addition, the positive and negative literals of each variable are grouped together so that they are adjacent in the total order used in constructing the ZBDD. This restriction facilitates, among other things, the identification and automatic removal of tautologies, i.e., combinations that have the form  $(x + x' + \dots)$ , to further reduce the size of the ZBDD [5].

To determine the size of the function represented by the CNF formula associated with a ZBDD node, we must first rewrite (10) as the disjoint sum of two terms

$$f = (g) \cdot (x + h) = x \cdot g + x' \cdot (g \cdot h). \quad (11)$$

This immediately leads to

$$\|f\| = \frac{1}{2}(\|g\| + \|g \cdot h\|) \quad (12)$$

which, unlike (6) for BDDs, requires that we compute the size of the product of the two child formulas. This is not a problem if one or both of the children is a terminal node, but does pose a serious complication if they are both internal nodes. We examined three solutions.

*Exact Computation:* One way to resolve this complication is to (recursively) create additional ZBDD nodes for such products until one of the children becomes terminal. This will provide us with the *exact* answer, but may exponentially increase the size of the ZBDD. Some of that increase can be ameliorated with caching and garbage collection. In particular, created nodes can be eliminated as soon as they have been used to compute the size of their parent.

*Approximate Computation:* An alternative to computing  $\|g \cdot h\|$  exactly is to *bound* it. The upper bound is easily established as  $\min(\|g\|, \|h\|)$  and occurs when either  $g \leq h$  or  $h \leq g$ . The lower bound can be determined by noting that  $\|g \cdot h\| = 1 - \|g' + h'\|$ . Thus,  $\|g \cdot h\|$  is smallest when  $\|g' + h'\|$  is largest which occurs when  $g'$  and  $h'$  are disjoint. This gives a lower bound of  $\|g\| + \|h\| - 1$  and yields the interval

$$\|g \cdot h\| \in [\max(0, \|g\| + \|h\| - 1), \min(\|g\|, \|h\|)] \quad (13)$$

where the max in the lower bound ensures that the estimate remains nonnegative.

An illustration of these computations is given in Fig. 2 for the example formula  $(a+b') \cdot (b+c)$ . The percentages annotating the ZBDD nodes denote the function sizes of their corresponding formulas as computed by (12) and (13). The uncertainty in the size at the top node is resolved, in Fig. 2(b), by creating a node for the product of its children.

*Controlled-Accuracy Computation:* Between the two extremes of an *exact* size and a *bound* computed according to (13)

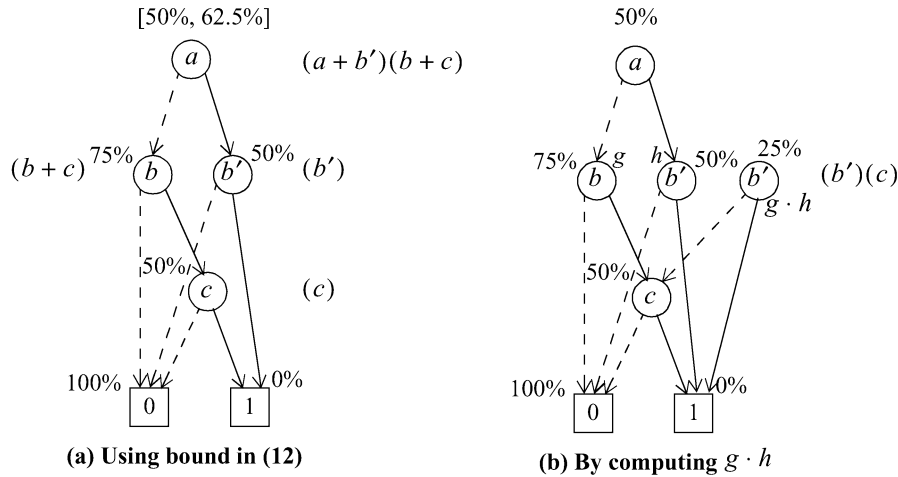


Fig. 2. Computation of  $\|(a + b') \cdot (b + c)\|$  using (12) and (13). Nodes are labeled by literals, and annotated with their associated formulas and their corresponding sizes.

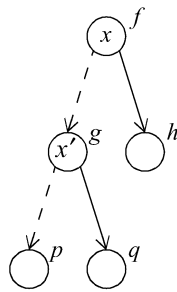


Fig. 3. The special case when  $g$  is not vacuous in  $x$ . Note that  $h$ 's node cannot be labeled by  $x'$  as this would create a tautology that is automatically eliminated.

we can produce a range of approximations that trade accuracy with speed and memory consumption. Specifically, when a given level of accuracy, say 10%, is exceeded by the bound computed from (13), additional ZBDD nodes are created for the product formulas until the desired level of accuracy is achieved.

We finally note that (12) is correct only when  $g$  is vacuous in  $x$ . The only situation when this is not true is depicted in Fig. 3 where  $g$ 's node is labeled by the literal  $x'$ . Substituting  $g = (p) \cdot (x' + q)$  in (11) produces the disjoint sum

$$f = x \cdot (p \cdot q) + x' \cdot (p \cdot h) \quad (14)$$

which readily leads to

$$\|f\| = \frac{1}{2}(\|p \cdot q\| + \|p \cdot h\|). \quad (15)$$

Fig. 4 illustrates the three possible computations on the bridging-fault *bf2670-001* instance [9].

## VI. EXPERIMENTAL EVALUATION

In this section, we show examples of how measuring search space coverage can be used to interpret the results of running SAT solvers as well as compare different heuristics. We conclude the section with an analysis of Satometer's performance.

Satometer is implemented in C++ using the CUDD package [23]. It incorporates the ZBDD enhancements described in [1]

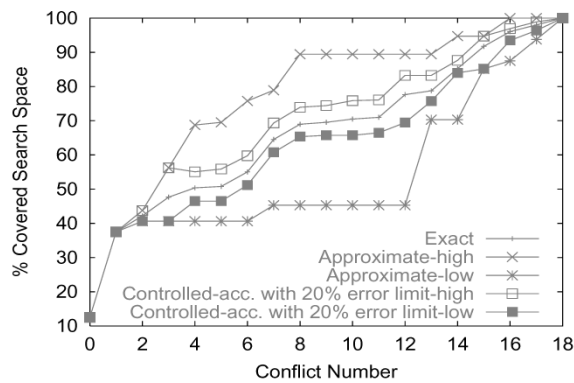


Fig. 4. Percentage of explored search space for the *bf2670-001* instance using the exact, approximate, and controlled-accuracy computations. Note that the approximate computation has a maximum error of 44% and the controlled-accuracy has a maximum error of 15%, even though the error limit was set to 20%.

and [5] for symbolic manipulation of CNF formulas. We configured it to report the size of the explored search space to within 20% of the exact answer; in many cases it was able to achieve a higher level of accuracy or to even report the exact answer. In the tables to follow, a single number in the “explored space” columns indicates that an exact answer was reported; ranges are indicated as intervals. All experiments were performed on an AMD Athlon 1.2-GHz machine with 500-MB RAM running the Linux operating system.

### A. Effect of Preprocessing the CNF Formula

A variety of preprocessing techniques have been proposed to modify a CNF formula before submitting it to a SAT solver. These techniques generally add clauses to the formula in order to increase the number of potential implications or perform stylized algebraic simplifications to reduce the number of variables. We used Satometer to measure the size of the search space covered by clauses obtained in preprocessing. We found an interesting relation between the size of the covered search space and the effectiveness of preprocessing, which does not indicate that the relation is always that simple. In each of the presented cases, we compared the size of the space explored by a standard DLL

TABLE II  
ADDITION OF SYMMETRY-BREAKING PREDICATES

Benchmark	Original		Modified	Explored Space, %	
	V	C	Extra C	Original	Modified
hole-7	56	204	14	100	100
hole-8	72	297	16	79.2	100
hole-9	90	415	18	37.5	100
hole-10	110	561	20	18.75	[99.98, 100]
hole-11	132	738	22	9.39	[99.96, 100]
hole-12	156	949	24	4.68	[99.96, 100]

TABLE III  
ALGEBRAIC SIMPLIFICATION

Benchmark	Original		Modified		Explored Space, %	
	V	C	V	C	Original	Modified
longmutl7	3319	10335	2184	7635	0.280	0.341
queinvar20	2435	20671	2343	28438	50	50.1
barrel7	3523	13765	800	3447	51.02	62.46
dlx2_cc_bug08	1515	12808	1486	13875	0	9.38

algorithm [8] (i.e., without conflict analysis) on the original as well as on the modified formula. The DLL solver uses a fixed decision heuristic, chronological backtracking, and implements BCP as in Chaff [15]. The time-out limit in these experiments was set to 10 s; *Satometer's run time was negligible*. The results of these experiments are given in Tables II and III.

*Addition of Symmetry-Breaking Predicates:* In [7], the authors propose analyzing a CNF formula: 1) to identify its symmetries, and 2) to augment it with clauses that break those symmetries. The intuition here is that the symmetry-breaking clauses act by allowing only one of many equivalent variable assignments to be a potential solution to the formula. If the original formula is satisfiable, the number of solutions may considerably decrease after preprocessing, clearly indicating that the search space was reduced. Yet, even if the original instance was not satisfiable, “the number of equivalent roads leading nowhere” would be reduced, and a generic SAT solver is likely to conclude much faster that no solution exists.

This intuition is confirmed by the data in Table II. Column 1 lists the name of the benchmark; columns 2 and 3 give the number of variables (V) and clauses (C) in the original formula; column 4 gives the number of symmetry-breaking clauses that are added to the formula; and columns 5 and 6 indicate the size of explored space reported by *Satometer*. The benchmarks in this experiment are members of the unsatisfiable *hole* suite (which relates to the Pigeonhole principle). The augmentation of each instance by a small number of symmetry-breaking clauses drastically enhances the ability of the SAT solver to prove unsatisfiability. This trend is clearly accentuated as instance sizes increase.

*Algebraic Simplification:* Another formula preprocessing technique is based on formula simplification rules aimed at reducing the number of variables or clauses in the formula [20]. We studied this approach on some large hard bounded model checking [3] and microprocessor verification [26] instances. Results on a representative sample are given in Table III.

Unlike the earlier experiments, the performance of the SAT solver on the modified formulas is not significantly better than its performance on the original formulas for the allotted amount of search time. The best improvement is in the *barrel7* benchmark and can be attributed to the simplifier’s ability to drastically reduce the number of variables (from 3523 to 800). Note that the addition of smaller clauses prunes more of the search space, but does not always accelerate finding a solution. In general, the harder it is to deduce a clause (by resolution), the greater is the effect of adding this clause to the current formula. For example, symmetry-breaking or conflict-induced clauses are valuable, due to the fact that deducing these clauses by resolution is expensive. *Satometer* can successfully measure the size of the search space pruned by such clauses, which typically involves overlapping portions of the search space.

### B. Analysis of Dynamic Techniques

In this set of experiments, we report on the application of *Satometer* to various SAT solvers with a variety of parameters. Our experiments involve three different SAT solvers: a simple DLL solver [8], SATIRE [27], and Chaff [15]. The last two solvers represent efficient implementations of the basic DLL solver. Chaff, however, is currently known as the leading DLL-based SAT solver. The goals of this set of experiments are to determine: 1) the best of two SAT solvers, in which each solver’s description is hidden; 2) the best of a variety of decision heuristics; 3) the amount of explored search space for difficult CNF instances; 4) the best of various conflict analysis techniques; and 5) an estimate of the number of satisfying assignments in a satisfiable instance.

*SAT Solver A Versus SAT Solver B Experiment:* In this experiment, several SAT solvers are provided. However, the user has no knowledge of the internals of any of the SAT solvers. Given a set of hard instances, the user is required to identify the best solver in the shortest possible time. In general, the user will need to run each SAT solver for a specified time or randomly select a solver and hope that it is the best among all others. Using the proposed method, however, can give an insight to which solver performs best within the specified run time limit. Table IV shows several results for various hard instances from bounded model checking [3], microprocessor verification [26], FPGA routing [16], and the DIMACS set [9]. We tested each instance for 10 s using the following three SAT solvers and options: standard DLL solver, Chaff with a fixed decision heuristic, and Chaff with the default cherry.smj heuristic. The results clearly indicate the superiority of the third solver for almost all benchmarks, due to the significantly high search space coverage achieved in the given time limit. Fig. 5(a)–(b) shows a detailed space coverage analysis of the *barrel5* instance for all three solvers.

*Comparison of Decision Heuristics:* As shown in Table IV, the proposed method can also be used to classify decision heuristics and rate their performance on various SAT instances. We show the results for two decision heuristics: 1) static fixed [9]: unresolved variables with minimum index are selected first for decisions; 2) dynamic VSIDS [15]: variables that appear in the highest number of clauses are selected first. (Some weight is given to variables appearing in recent conflict-induced clauses). Again, the results show the effectiveness of VSIDS as opposed

TABLE IV  
PERCENTAGE OF EXPLORED SEARCH SPACE FOR VARIOUS SAT SOLVERS AND DECISION HEURISTICS. THE SEARCH RUN TIMES USING THE CHAFF [15] SAT SOLVER ARE ALSO LISTED

Benchmark					Space Explored,%			Chaff-VSIDS
Family	Name	S/U	V	C	DLL	Chaff-Fixed	Chaff-VSIDS	Search Time (sec)
uP Verification	2dlx_cc	U	4524	41704	0	[81, 100]	[99.06, 100]	17
	3pipe	U	2392	27533	0.098	[47.23, 62.63]	[80.41, 100]	50
	4pipe	U	5096	80213	0.025	[69.68, 88.15]	[77.77, 95.46]	334
	9vliw	U	19148	179492	0	[28.91, 35.16]	[99.94, 100]	616
DIMACS	par32-1-c	S	1315	5254	0	[78.64, 89.39]	[82.72, 100]	>10000
Bounded Model Checking	barrel6	U	2306	8931	52.77	[60.94, 63.83]	100	9.5
	barrel7	U	3523	13765	51.02	[60.95, 68.79]	[98.34, 100]	70
	barrel9	U	8903	36606	50.11	[58.59, 58.84]	[99.94, 100]	610
	longmult6	U	2848	8853	0.40	[72.39, 80.43]	[99.93, 100]	10.2
	longmult8	U	3810	11877	0.21	[80.27, 87.78]	[90.48, 100]	125
	queuein18	U	2081	17368	0	[96.57, 100]	100	1.6
	queuein20	U	2435	20671	50	[92.3, 100]	[97.59, 100]	11.1
FPGA Routing	alu2_gr_rcs_w7	U	3570	73478	2.36	[29.99, 36.55]	[50, 58.75]	12.3
	k2fix_gr_rcs_w8	n/a	10056	271393	1.18	[0.665, 7.65]	[0.798, 9.03]	>10000
	k2fix_gr_rcs_w9	n/a	11313	305160	0.59	[0.393, 5.147]	[0.400, 3.34]	>10000
	vda_gr_rcs_w8	S	5776	116522	0	[0.615, 6.65]	[0.819, 9.75]	>10000

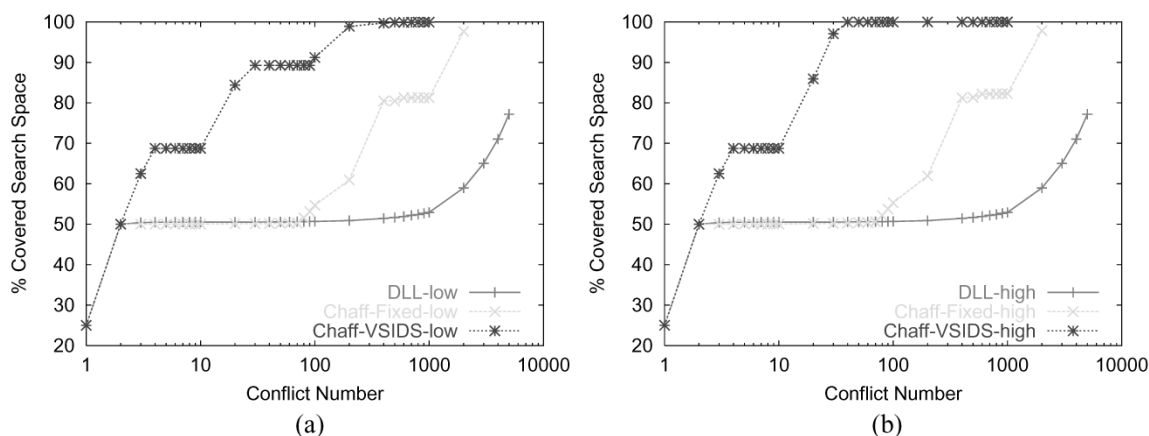


Fig. 5. Search space coverage for *Barrel5* instance. (a) Lower and (b) upper bounds are shown.

to the fixed decision heuristic within the specified run time limit. Nevertheless, the *k2\_fix\_gr\_rcs\_w9* instance shows a larger upper bound of the explored search space using the fixed decision order as opposed to VSIDS. However, since the ranges for both heuristics overlap, it is hard to identify the optimal decision heuristic.

*Postmortem on Difficult Instances:* Table IV also shows the small percentage of explored search space for the FPGA routing instances for the given decision heuristics and SAT solvers. Fig. 6(a) shows a detailed space coverage analysis of the *k2\_fix\_gr\_rcs\_w9* [16] instance after unsuccessfully trying to solve it with Chaff for up to 500 s. The fact that more than 55% of the search space is still not explored *might* suggest a hard problem instance. In fact, the instance was still unsolved after running Chaff for 10 h. Fig. 6(b) shows the space coverage analysis of the *9\_vliw\_bp\_mc* instance after unsuccessfully trying to solve it with Chaff for up to 500 s. Since most of the search space is explored, this *might* suggest an instance with

a few satisfying assignments or an unsatisfiable instance. The instance was proven to be unsatisfiable. Note that the size of the covered search space does not correlate with the complexity of exploring the rest of the search space. For example, Chaff was successfully able to explore, after running for 10 s, more than 99.9% of the search space for the *barrel6* and *longmult6* instances. Yet, it solved the instances in 10.1 and 610 s, respectively.

*One UIP Versus All UIPs Conflict Analysis:* Recently, an analysis of various conflict-induced clause learning schemes was reported in [28]. The authors found that different learning schemes can significantly affect the behavior of SAT solvers. Based on various EDA instances, they were able to demonstrate that the learning scheme based on the first unique implication point (UIP) [21] of the implication graph is more effective in solving SAT problems than other schemes such as the “All UIP” approach. In order to further confirm this conclusion, we plotted the growth range, using the “All UIP,” “1 UIP,” and “0 UIP”

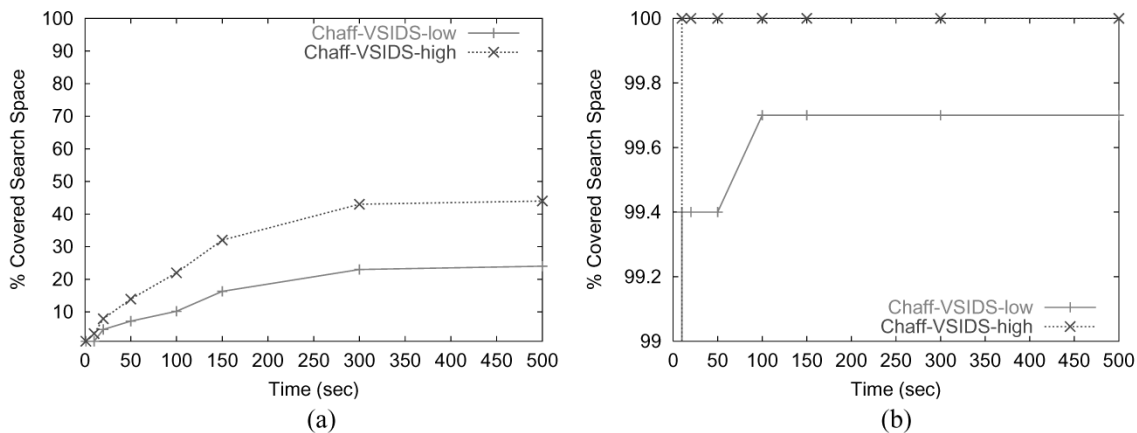


Fig. 6. Search space coverage for the (a) *k2\_fix\_gr\_rcs\_w9* and (b) *9\_vliw\_bp\_mc* instances.

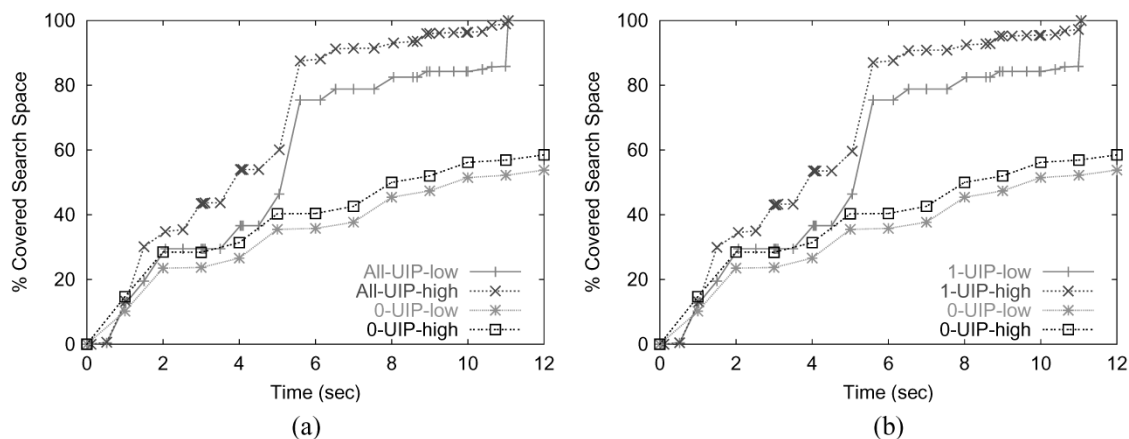


Fig. 7. Search space coverage of the unsatisfiable *queueinvar8* instance using (a) the All UIP versus (b) 1 UIP conflict analysis learning schemes. The plots also show the search space coverage when learning is disabled (0 UIP).

(clause learning is disabled) approaches for the *queueinvar8* instance from the bounded model checking set [3]. We implemented all three approaches in SATIRE. Fig. 7 shows the runs using the SATIRE SAT solver. The coverage percentage was measured after each backtrack call. As the plot clearly shows, the addition of all UIPs resulted in a minor benefit and perhaps slowed the search process as additional time is spent to generate all the UIP clauses. Furthermore, the instance was unsolved after running SATIRE with disabled clause learning for 12 s. This detailed analysis of the internals of the search process can provide a better understanding of a problem's structure and the effectiveness of the SAT solver and enhancement being tested.

*Number of Satisfiable Assignments:* As mentioned earlier, the search space will never be totally explored in "satisfiable" instances as SAT solvers typically exit after identifying the first satisfying assignment. However, in some cases several satisfying assignments, if not all, are needed. An example is to identify all possible primary input assignments for a circuit that would minimize the total gate delay. An insight into the number of possible satisfying assignments can be very helpful. A satisfiable instance in which a satisfying assignment is identified at an early stage of the search process is likely to have many satisfying assignments. In contrast, an instance that identifies a satisfying assignment after exploring almost the complete search space probably has few satisfying assignments. In order to test our assumption, we selected two satisfiable instances from the DIMACS set [9],

TABLE V  
PERCENTAGE OF EXPLORED SEARCH SPACE FOR SATISFIABLE INSTANCES WITH DIFFERENT NUMBERS OF SATISFYING ASSIGNMENTS

Benchmark	Explored Space, %
aim-200-1_6-yes1-1.cnf	99.999
ssa7552-160.cnf	28.125

namely the *aim-200-1\_6-yes1-1* and *ssa7552-160*. The former is known to have a single satisfying assignment, whereas the latter represents a stuck-at-fault problem with many satisfying assignments. Both instances were solved by Chaff in less than a second. We measured the explored search space after the search was completed for a single satisfying assignment. Table V shows the results. As expected, the percentage of the search space explored for the *aim\** instance was tremendously larger than that of the *ssa\** instance.

Again, as in the experiments in Section VI-A, the accuracy of these results is significant. Although a user-specified error limit of 20% is set, out of the 78 runs, 47, 6, 16, 8, reported results with 100%, > 99%, 90% ~ 99%, 80% ~ 90% accuracy.

In terms of run time and memory consumption, constructing the ZBDDs is fast and is usually dependent on the size of the clauses. Furthermore, the high compression power of the ZBDD data structure utilizes less memory than a list data structure [1], [5], [14]. As mentioned in Section V-B, computing the search



TABLE VI  
ANALYSIS OF VARIOUS COVERAGE COMPUTATION TECHNIQUES IN SATOMETER. EACH INSTANCE WAS TESTED FOR 10 s USING CHAFF. SATOMETER’S RUN TIME AND MEMORY LIMITS WERE SET TO 100 s AND 500 MB OF RAM. “TIME-OUT” AND “MEM-OUT” INDICATE TECHNIQUE ABORTED DUE TO EXCEEDING RUN TIME AND MEMORY LIMITS, RESPECTIVELY

Coverage Computation Technique		Benchmark Name							
		4pipe		par32-1-c		barrel9		k2fix_gr_rcs_w9	
		Satometer Time	Explored Space, %	Satometer Time	Explored Space, %	Satometer Time	Explored Space, %	Satometer Time	Explored Space, %
BDD	without sifting	mem-out	--	mem-out	--	mem-out	--	mem-out	--
	with sifting	time-out	--	time-out	--	time-out	--	time-out	--
ZBDD-based Satometer Controlled Accuracy	0% (exact)	time-out	--	time-out	--	time-out	--	time-out	--
	5%	time-out	--	5.35	[95.51, 100]	0.49	[99.9, 100]	0.69	[0.4, 3.34]
	10%	time-out	--	0.99	[90.58, 100]	0.19	[99.9, 100]	0.69	[0.4, 3.34]
	20%	<b>0.28</b>	<b>[77.7, 95.46]</b>	<b>0.72</b>	<b>[82.72, 100]</b>	<b>0.19</b>	<b>[99.9, 100]</b>	<b>0.58</b>	<b>[0.4, 3.34]</b>
	50%	0.28	[62.5, 99.99]	0.66	[50.78, 100]	0.2	[99.9, 100]	0.53	[0.4, 3.34]
	100% (approx.)	0.27	[62.5, 99.99]	0.66	[50.78, 100]	0.2	[99.9, 100]	0.53	[0.4, 3.34]

space coverage with an unrestricted bound is done by a single traversal of the ZBDD. On the other hand, the restricted bound and the exact count methods are slower, since additional ZBDD nodes are created during the ZBDD traversal. The size of the ZBDD, however, does not grow exponentially since the additional ZBDD nodes are removed as soon as the function sizes of their corresponding formulas are computed.

One way to reduce the run time and memory consumption is to only analyze conflict-induced clauses of size  $k$  or less. In general, smaller clauses are more useful in measuring the explored search space and require less ZBDD construction time and fewer ZBDD nodes. This approach, however, can only be used to measure the lower bound of the explored search space. For the instances reported in Tables IV and V, Satometer was able to compute the search space coverage for almost all instances in less than a second each.

In order to further analyze the performance of Satometer, we selected four instances from each family of benchmarks in Table IV. Satometer was applied with various accuracy settings to all four instances, after testing each instance for 10 seconds using Chaff with the default cherry.smj heuristic (i.e., the dynamic VSIDS decision heuristic.) The results are shown in Table VI. The table also shows a comparison between using ZBDDs and BDDs to compute the space coverage. The presented techniques were configured to analyze conflict-induced clauses consisting of 50 literals or less. A cursory analysis of the data suggests several observations.

- 1) The BDD-based approach without sifting [18], [23], quickly runs out of memory for all instances. The use of dynamic sifting reduces the memory usage but requires longer run times to compute the space coverage. In fact, the use of sifting with the BDD approach times out for all four instances.
- 2) Representing CNF formulas with ZBDDs, on the other hand, can be done with modest memory requirements and faster manipulation run times. Although sifting is dis-

abled, ZBDDs never run out of memory for any of the presented instances. Sifting is disabled in all of the presented experiments, since it slows down Satometer’s performance.

- 3) The use of the “exact” computation times out for all instances. This is mainly due to the addition of a significant number of ZBDD nodes during the computation of the space coverage.
- 4) The use of the “approximate” computation is the fastest among all presented approaches, since a single traversal of the ZBDD is required to compute the search space coverage. The bound, however, can be inaccurate. For example, the *4-pipe* and *par32-1-c* instances have errors of 37.5% and 49%, respectively.
- 5) Varying the “controlled accuracy” limit allows a trade-off between accuracy and speed. In general, smaller error limits require longer computation run times but yield accurate results. The use of higher error limits speeds up the computation, but can yield inaccurate results. Applying Satometer with a 20% error limit successfully computed the space coverage for all instances within reasonable run times.
- 6) Our final observation is that even when a high error limit is used, e.g., the approximate computation, two instances report results with almost 95% accuracy.

## VII. SUMMARY AND CONCLUSION

We described Satometer, a tool that measures the percentage of search space explored by a SAT solver. The tool can provide helpful diagnostic information, either during or at the conclusion of a SAT run. We believe that tools such as this are needed to complement the powerful SAT engines that have been developed in recent years. We plan to identify other metrics that can help characterize a search process (e.g., the maximum number of satisfied clauses encountered at any point during the search), to look

for ways to further improve the efficiency of Satometer (e.g., by caching computation results), and to use it to analyze the performance of solvers on hard SAT instances. We are also planning to integrate Satometer into known SAT solvers and use the search space information to improve decision and restart heuristics.

## REFERENCES

- [1] F. Aloul, M. Mneimneh, and K. Sakallah, "Backtrack search using ZBDD's," in *Proc. Int. Workshop Logic Synthesis (IWLS)*, 2001, pp. 293–297.
- [2] R. Bayardo Jr. and R. Schrag, "Using CSP look-back techniques to solve real world SAT instances," in *Proc. 14th National Conf. Artificial Intelligence (AAAI)*, 1997, pp. 203–208.
- [3] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDD's," in *Proc. Design Automation Conf. (DAC)*, 1999, pp. 317–320.
- [4] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, pp. 677–691, Aug. 1986.
- [5] P. Chatalic and L. Simon, "Multi-resolution on compressed sets of clauses," in *Proc. Int. Conf. Tools Artificial Intelligence*, 2000, pp. 2–10.
- [6] P. C. Chen, "Heuristic sampling: A method for predicting the performance of tree searching programs," *SIAM J. Comput.*, vol. 21, no. 2, pp. 295–315, 1992.
- [7] J. Crawford, M. Ginsberg, E. Luks, and A. Roy, "Symmetry-breaking predicates for search problems," *Knowledge Represent.: Principles Knowledge Represent. Reason.*, pp. 148–159, 1996.
- [8] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, 1962.
- [9] DIMACS Challenge Benchmarks [Online]. Available: <ftp://Dimacs.rutgers.EDU/pub/challenge/sat/benchmarks/cnf>.
- [10] D. E. Knuth, "Estimating the efficiency of backtrack programs," *Math. Comput.*, vol. 29, pp. 121–136, 1975.
- [11] D. Kokotov and I. Shlyakhter. (2000) Progress Bar for SAT Solvers. [Online]. Available: <http://sdg.lcs.mit.edu/satsolvers/progressbar.html>
- [12] T. Larrabee, "Test pattern generation using Boolean satisfiability," *IEEE Trans. Computer-Aided Design*, vol. 11, pp. 4–15, Jan. 1992.
- [13] M. Löbbling, O. Schröder, and I. Wegner, "The theory of zero-suppressed BDD's and the number of knight's tours," *Formal Meth. Syst. Design*, vol. 13, pp. 235–253, 1995.
- [14] S. Minato, "Zero-suppressed BDD's for set manipulation in combinatorial problems," in *Proc. Design Automation Conf. (DAC)*, 1993, pp. 272–277.
- [15] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proc. Design Automation Conf. (DAC)*, 2001, pp. 530–535.
- [16] G. Nam, F. Aloul, K. Sakallah, and R. Rutenbar, "A comparative study of two Boolean formulations of FPGA detailed routing constraints," in *Proc. Int. Symp. Physical Design*, 2001, pp. 222–227.
- [17] P. Purdom, "Tree size by partial backtracking," *SIAM J. Computing*, vol. 7, no. 4, pp. 481–491, 1978.
- [18] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," presented at the Int. Conf. Computer Aided Design (ICCAD), San Jose, CA, 1993.
- [19] B. Selman, H. Kautz, and B. Cohen, "Noise strategies for local search," in *Proc. 11th National Conf. Artificial Intelligence*, 1994, pp. 337–343.
- [20] J. Silva, "Algebraic simplification techniques for propositional satisfiability," presented at the Int. Conf. Principles Practice Constraint Programming, Singapore, 2000.
- [21] J. Silva and K. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Trans. Comput.*, vol. 48, pp. 506–521, May 1999.
- [22] L. Silva, J. Silva, L. Silveira, and K. Sakallah, "Timing analysis using propositional satisfiability," *Proc. IEEE Int. Conf. Electronics, Circuits Systems*, vol. 3, pp. 95–99, 1998.
- [23] F. Somenzi. CUDD: CU Decision Diagram Package. Univ. Colorado at Boulder, Boulder, CO. [Online]. Available: <ftp://vlsi.colorado.edu/pub/>
- [24] G. Stålmarck, "System for determining propositional logic theorems by applying values and rules to triplets that are generated from Boolean formula," Swedish Patent 467 076, 1994.
- [25] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, "Combinational test generation using satisfiability," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 1167–1176, Sept. 1996.
- [26] M. Velev and R. Bryant, "Boolean satisfiability with transitivity constraints," in *Proc. Conf. Computer-Aided Verification (CAV)*, 2000, pp. 86–98.
- [27] J. Whitemore, J. Kim, and K. Sakallah, "SATIRE: A new incremental satisfiability engine," in *Proc. Design Automation Conf. (DAC)*, 2001, pp. 542–545.
- [28] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik, "Efficient conflict driven learning in a Boolean satisfiability solver," in *Proc. Int. Conf. Computer Aided Design (ICCAD)*, 2001, pp. 279–285.
- [29] H. Zhang, "SATO: An efficient propositional prover," in *Proc. Int. Conf. Automated Deduction*, 1997, pp. 272–275.



**Fadi A. Aloul** (S'02) received the B.S. degree in electrical engineering *summa cum laude* from Lawrence Technological University, Southfield, MI, in 1997 and the M.S. and Ph.D. degrees in computer science and engineering from the University of Michigan, Ann Arbor, in 1999 and 2003, respectively.

He is currently a post-doc Research Fellow at the University of Michigan. His current research interests are in the areas of computer-aided design, verification, and Boolean satisfiability.

Dr. Aloul is currently the AV Chair of the 2003 International Workshop on Logic Synthesis (IWLS). He is also serving on the technical committee of the International Workshop on Logic Synthesis, the International Conference on Theory and Applications of Satisfiability Testing (SAT), and the International Workshop on Soft Constraints. He has received a number of awards, including the Agere/SRC research fellowship, GANN fellowship, and the LTU presidential scholarship.



**Brian D. Sierawski** received the B.S.E degree in computer engineering from the University of Michigan, Ann Arbor, in 2002. He is currently working toward the Ph.D. degree at the same university.

In 2001, he was an Intern with Intel, Portland, OR. Since 2000, he has been working at the Advanced Computer Architecture Laboratory, the University of Michigan, studying the verification of complex systems.



**Karem A. Sakallah** (S'76–M'81–SM'92–F'98) received the B.E. degree in electrical engineering from the American University of Beirut, Beirut, Lebanon, and the M.S.E.E. and Ph.D. degrees in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA, in 1975, 1977, and 1981, respectively.

In 1981, he was with the Department of Electrical Engineering at Carnegie Mellon University as a Visiting Assistant Professor. From 1982 to 1988, he was with the Semiconductor Engineering Computer-Aided Design Group at Digital Equipment Corporation, Hudson, MA, where he headed the Analysis and Simulation Advanced Development Team. Since September 1988, he has been with the University of Michigan, Ann Arbor, as a Professor of Electrical Engineering and Computer Science. From September 1994 to March 1995, he was with the Cadence Berkeley Laboratory in Berkeley, CA, on a six-month sabbatical leave. He has authored or coauthored more than 150 papers and has presented seminars and tutorials at many professional meetings and various industrial sites. His research interests include the area of computer-aided design with emphasis on simulation, timing verification and optimal clocking, logic and layout synthesis, Boolean satisfiability, and design verification.

Dr. Sakallah was an Associate Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS from 1995 to 1997, and has served on the Program Committees of the International Conference on Computer-Aided Design, Design Automation Conference, and the International Conference of Computer Design as well as and numerous other workshops. He is currently an Associate Editor of the IEEE TRANSACTIONS ON COMPUTERS. He is a Member of the Associate of Computing Machinery (ACM) and Sigma Xi.