Satometer: How Much Have We Searched?

Fadi A. Aloul, Brian D. Sierawski, Karem A. Sakallah
Department of Electrical Engineering and Computer Science
University of Michigan
{faloul, bsieraws, karem}@eecs.umich.edu

ABSTRACT

We introduce Satometer, a tool that can be used to estimate the percentage of the search space actually explored by a backtrack 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 BDD 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.

Categories and Subject Descriptors

I.1 [**Symbolic and Algebraic Manipulation**]: Expressions and Their Representation, Algorithms.

General Terms

Algorithms, Measurement, Experimentation, Verification.

Keywords

BDDs, ZBDDs, SAT, CNF, backtrack search, conflict diagnosis, search space coverage, search progress.

1. INTRODUCTION

The last few years have seen significant algorithmic advances in, and carefully-crafted implementations of, Boolean Satisfiability (SAT) solvers [3, 13, 15, 19]. This has led to their successful application to a wide range of large-scale EDA problem instances consisting of thousands of variables and millions of clauses [4, 16, 18]. 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.

The paper is organized as follows. In Section 2 we introduce our measure of search progress. We then describe, in Section 3, how this measure can be computed using BDDs and ZBDDs. In Section 4 we illustrate the utility of this measure in a variety of experimental scenarios and conclude, in Section 5, with a summary of the paper's main contributions.

2. MEASURING SEARCH PROGRESS

Despite the considerable activity in SAT research, the question of measuring the progress of a search process does not seem to have attracted much attention. The only relevant work that we were able to find is that of Kokotov and Shlyakhter [10]. They describe a progress bar that can be integrated into a backtrack SAT solver to measure its progress. The bar is updated based on either historical or predictive estimates of the size of the decision tree maintained by the SAT solver. They reported that the bar is able to predict progress with an accuracy of 80-90% without significantly impacting the solver's run time.

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 having been *explored* and found to contain no solutions.

Let $A_1, A_2, \ldots, A_i$ 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[1] covered by the function $A_1 + A_2 + \ldots + A_i$. 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 $\|a \oplus b\| = 50\%$ [2]. In the sequel, we will refer to $\| f \|$ as the *size* of f.

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

1. Complete truth assignment that sets the function to 1.
2. Assuming that the number of variables is 2.

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

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

(a) Execution trace of a basic backtrack SAT Solver

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

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

Figure 1. Execution traces of two different SAT solvers on the formula in (1) illustrating how search progress is measured.

An illustration of these computations is shown in Figure 1 for the 4-variable formula

$$\varphi = (a+b+c)(a+b+c')(a'+b+c')(a+c+d)\cdot (a'+c+d)(a'+c+d')(b'+c'+d')(b'+c'+d) \quad (1)$$

3. COMPUTING 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:

$$\|A_1 + A_2 + \ldots + A_i\| = \sum_{1 \le k \le i} \|A_k\| \quad (2)$$

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

$$1 - \|C_1 \cdot C_2 \cdot \ldots \cdot C_j\| = \sum_{1 \le k \le j} (1 - \|C_k\|) \quad (3)$$

In other words, if conflicts identify non-overlapping pieces of the search space, then the size of the explored space can be found by simply adding the sizes of the different pieces. In general, this will not be the case except for standard backtrack algorithms that do not employ conflict diagnosis to prune the search space. To compute the size of the explored space in such cases we have no choice but to build some type of symbolic representation for the disjunction of conflict assignments or the conjunction of conflict clauses. We describe below the 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 clauses.

3.1 Using BDDs

The conflict clauses can be symbolically "anded" using a reduced ordered binary decision diagram (ROBDD or BDD for short) [5]. BDD semantics allow us to write the function f at a node labeled with variable x using Boole's expansion:

$$f = x' \cdot g + x \cdot h \quad (4)$$

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

$$\|f\| = \frac{1}{2}(\|g\| + \|h\|) \quad (5)$$

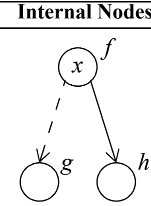
The size of the function represented by a BDD can now be obtained by sweeping the BDD from the terminal nodes towards the top node and applying (5) at each visited node. The sweep is initialized by setting $\|0\| = 0$ and $\|1\| = 1$ for the constant functions of the terminal nodes.

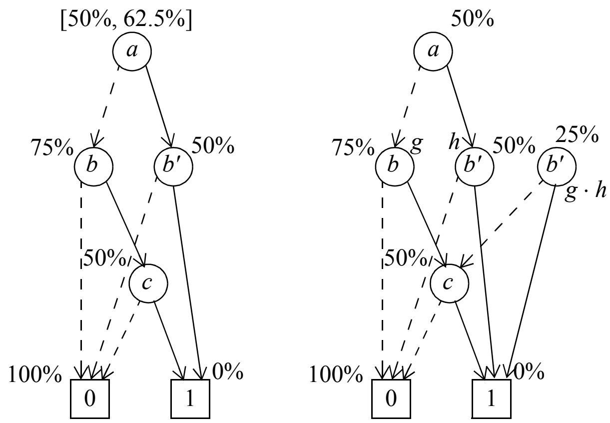
3.2 Using ZBDDs

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 zero-suppressed BDD (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 [6] demonstrated that ZBDDs can be an effective implicit representation of large CNF formulas and showed how they can be used to perform "multi-resolution" to solve some large structured SAT instances. In this scenario, the above example set corresponds to the CNF formula $(a+b)(c+d+e)(a+d)(b)$, 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. $(a)(a'+b+c) = (a)(b+c)$, were subsequently described in [1].

The semantics of ZBDD nodes were first articulated by Lobbing et al. in [11]. Given a set of atoms $\{a, b, c, \ldots\}$, a ZBDD node labeled with atom x represents a combination set f constructed according to the formula:

Table 1. Semantics of Decision Diagrams

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



(a) Using bound in (12) **(b) By computing $g \cdot h$**

Figure 2. Computation of $\|(a+b')(b+c)\|$ using (11) and (12).

$$f = g \cup \{x\} \times h \tag{6}$$

where g and h are the combination sets associated with the 0- and 1-children of that node (see Table 1.) 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 (6) is similar to the Cartesian product of two sets and is defined by

$$S \times T = \bigcup_{s \in S,\, t \in T} \{s \cup t\} \tag{7}$$

For example, given the combination sets $S = \{\{a, b\}, \{b, c\}\}$ and $T = \{\{a, d\}, \{e\}\}$, their product is[3]

$$S \times T = \{\{a, b, d\}, \{a, b, e\}, \{a, b, c, d\}, \{b, c, e\}\} \tag{8}$$

When used to represent a CNF formula, the formula f associated with a ZBDD node labeled by variable x follows the same template of (6) 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

$$f = (g) \cdot (x + h) \tag{9}$$

where g and h are the formulas associated with the 0- and 1-children of that node (see Table 1.) 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 $(x + x' + \ldots)$, to further reduce the size of the ZBDD [6].

To determine the size of the function represented by the CNF formula associated with a ZBDD node, we must first re-write (9) as the disjoint sum of two terms

$$f = (g) \cdot (x + h) = x \cdot g + x' \cdot (g \cdot h) \tag{10}$$

This immediately leads to

$$\|f\| = \frac{1}{2}(\|g\| + \|g \cdot h\|) \tag{11}$$

which, unlike (5) 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. 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 tighten the bound of their parent.

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 $g \le h$ or $h \le g$. 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

$$\|g \cdot h\| \in [\max(0, \|g\| + \|h\| - 1), \min(\|g\|, \|h\|)] \tag{12}$$

where the max in the lower bound insures that the estimate remains non-negative.

An illustration of these computations is given in Figure 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 (11) and (12). The uncertainty in the size at the top node is resolved, in part b of the figure, by creating a node for the product of its children.

Between the two extremes of an *exact* count and a *bound* computed according to (12) 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 (12), additional ZBDD nodes are created for the product formulas until the desired level of accuracy is achieved.

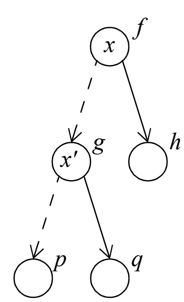


Figure 3. The special case when g is not vacuous in x.

We must finally note that (11) is correct only when g is vacuous in x. The only situation when this is not true is depicted in Figure 3 where g's node is labeled by the literal x'.[4] Substituting $g = (p) \cdot (x' + q)$ in (10) produces the disjoint sum

$$f = x \cdot (p \cdot q) + x' \cdot (p \cdot h) \tag{13}$$

which readily leads to

$$\|f\| = \frac{1}{2}(\|p \cdot q\| + \|p \cdot h\|) \tag{14}$$

Figure 4 illustrates the three possible modes of our approach on the bridging-fault *bf2670-001* instance. Despite setting an error limit of 20%, on average, the restricted bound and the unrestricted bound methods reported results within 7% and 24%, respectively, of the exact answer.

4. EXPERIMENTAL EVALUATION

Satometer is implemented in C++ using the CUDD package [17]. It incorporates the ZBDD enhancements described in [1] and [6] for symbolic manipulation of CNF formulas. In this section we demon-

3. Note that $S \times T \neq S \cup T$. For this example, $S \cup T = \{\{a, b\}, \{b, c\}, \{a, d\}, \{e\}\}$.

4. Note that h's node cannot be labeled by x' as this would create a tautology that is automatically eliminated.

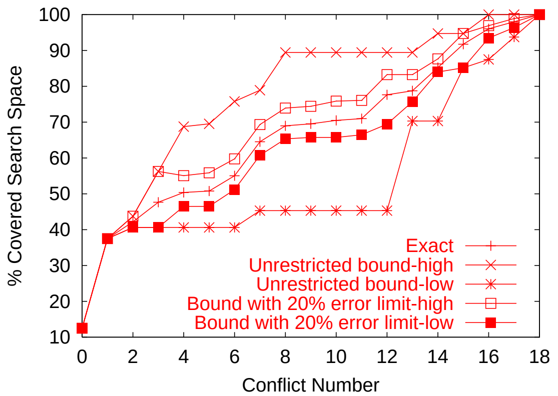


Figure 4. Applying three modes of the proposed approach to *bf2670-001.cnf* instance

strate its utility by applying it in a number of experimental scenarios. 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 1.4 GHz machine with 1GB of RAM running the Linux operating system.

4.1 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 study the effectiveness of such techniques. In each case we compare the size of the space explored by a standard DLL algorithm[5] [8] (i.e. without conflict analysis) on the original as well as on the modified formula. The time-out limit in these experiments was set to 10 seconds; *Satometer's run time was negligible*. The results of these experiments are given in Table 2, Table 3, and Table 4.

Addition of consensus clauses. In [2] the authors report that augmenting a CNF formula with clauses identified using consensus can reduce search time. To avoid generating an exponential number of clauses, they proposed a truncated iterative consensus procedure that augments the original formula with clauses whose size (number of literals) is limited by a small user-specified constant. They report speedups on the *aim* benchmarks from the DIMACS set [9] when the size of added clauses are limited to 3 or fewer literals.

A sampling of results on some unsatisfiable instances from this suite is shown in Table 2. 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 consensus clauses that are added to the formula; and columns 5 and 6 indicate the size of explored space reported by Satometer. The data in this table clearly show the effectiveness of these added clauses. For the two smaller instances, the search algorithm was actually able to explore the entire search space, and thus prove the unsatisfiability of the

5. The solver uses a fixed decision heuristic, chronological backtracking, and implements BCP as implemented in Chaff.

Table 2. Addition of consensus clauses

Benchmark	Original		Modified	Explored Space, %	
	V	C	Extra C	Original	Modified
aim-50-1_6-no-4	50	80	54	57.06	100
aim-100-1_6-no-3	100	160	73	0.015	100
aim-200-1_6-no-3	200	320	233	0.049	[97.72, 100]
aim-200-2_0-no-1	200	400	191	0	[87.75, 100]

Table 3. 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 4. 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

modified formula. In all cases, the addition of these clauses helped the SAT solver explore a significantly larger portion of the search space in the allotted amount of time.

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. However, 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 3 (whose layout is identical to that of Table 2.) The benchmarks in this experiment are members of the unsatisfiable *hole* suite [9] (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 [12]. We studied this approach on some large hard bounded model checking [4] and microprocessor verification [18] instances. Results on a representative sample are given in Table 4.

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. 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.)

Table 5. Percentage of Explored Search Space for Various SAT Solver and Decision Heuristics

Benchmark				Space Explored,%		
Family	**Name**	**V**	**C**	**DLL**	**Chaff–Fixed**	**Chaff–VSIDS**
uP Verification	**2dlx_cc**	4524	41704	0	[81, 100]	[99.06, 100]
	3pipe	2392	27533	0.098	[47.23, 62.63]	[80.41, 100]
	4pipe	5096	80213	0.025	[69.68, 88.15]	[77.77, 95.46]
	9vliw	19148	179492	0	[28.91, 35.16]	[99.97, 100]
DIMACS	**par32-1-c**	1315	5254	0	[78.64, 89.39]	[82.72, 100]
Bounded Model Checking	**barrel6**	2306	8931	52.77	[60.94, 63.83]	100
	barrel7	3523	13765	51.02	[60.95, 68.79]	[98.34, 100]
	barrel9	8903	36606	50.11	[58.59, 58.84]	[99.94, 100]
	longmult6	2848	8853	0.40	[72.39, 80.43]	[99.93, 100]
	longmult8	3810	11877	0.21	[80.27, 87.78]	[90.48, 100]
	queuin18	2081	17368	0	[96.57, 100]	100
	queuin20	2435	20671	50	[92.3, 100]	[97.59, 100]
FPGA Routing	**alu2_gr_rcs_w7**	3570	73478	2.36	[29.99, 36.55]	[50, 58.75]
	k2fix_gr_rcs_w8	10056	271393	1.18	[0.665, 7.65]	[0.798, 9.03]
	k2fix_gr_rcs_w9	11313	305160	0.59	[0.393, 5.147]	[0.400, 3.34]
	vda_gr_rcs_w8	5776	116522	0	[0.615, 6.65]	[0.819, 9.75]

The low coverage in this experiment is also an indication of the difficulty of these instances.

4.2 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 two different SAT solvers: a simple DLL solver [8] and Chaff [15]. Chaff represents an efficient implementation of the basic DLL solver and is currently known as the leading DLL-based SAT solver. The goals of this experiment are to determine a) the best of two black-box SAT solvers, in which each solver's description is hidden, b) the best of a variety of decision heuristics, c) the difficulty of CNF instances, and d) an estimate of the number of satisfying assignments in a satisfiable instance.

Black box A vs. black box B experiment. In the following 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 5 shows several results for various hard instances from bounded model checking [4], microprocessor verification [18], FPGA routing [16], and the DIMACs set [9]. We tested each instance for 10 seconds 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 among the other two solvers for almost all benchmarks, due to the significantly high search space coverage achieved in the given time limit. Figure 5 shows a detailed space coverage analysis of the *barrel5* instance for all three solvers.

Comparison of decision heuristics. As shown in Table 5, 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: a) static fixed [8]: unresolved variables with minimum index are selected first for decisions; b) 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 to the fixed decision heuristic. Nevertheless, the *k2fix_gr_rcs_w9* instance show 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, its hard to identify the optimal decision heuristic.

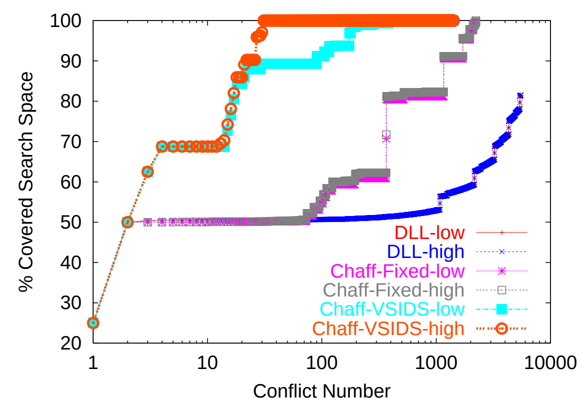


Figure 5. Search Space Coverage for *Barrel5.cnf*

Hard problem prediction. Table 5 also shows the difficulty of solving the FPGA routing instances as opposed to other hard instances for the given decision heuristics and SAT solvers. Figure 6 shows a detailed space coverage analysis of the *k2fix_gr_rcs_w9* instance after unsuccessfully trying to solve it with Chaff for up to 150 seconds. Perhaps, this method can be used as a metric to rate the difficulty of SAT instances and assist SAT solver developers in improving their SAT tools.

Number of satisfiable assignments. As mentioned earlier, the search space will never be totally explored in "satisfiable" instances, as SAT solvers typically abort 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,

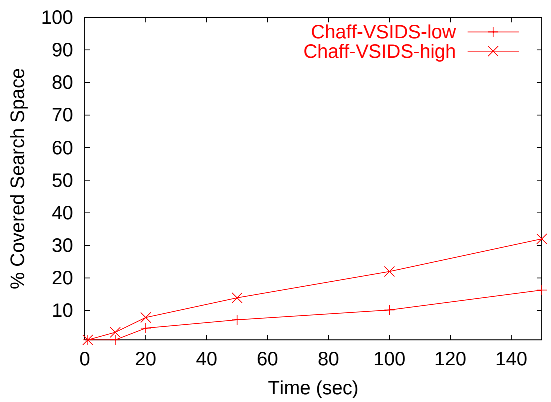


Figure 6. Search Space Coverage for *k2fix_gr_rcs_w9.cnf*

an instance that identifies a satisfying assignment after exploring almost the complete search space probably has a few satisfying assignments only. In order to test our assumption, we selected two satisfiable instances from the DIMACS set [9], namely the *aim-200-1_6-yes1-1.cnf* and *ssa7552-160.cnf*. The former is known to have a single satisfying assignment only, 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. As expected, the percentage of the search space explored by the *aim** instance (99.99%) was tremendously larger than the *ssa** instance (28.13%).

Again, as in the experiments in Section 4.1, the accuracy of our results are 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. As mentioned in Section 3.2, computing the search 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, doesn't 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 Table 5, Satometer was able to compute the search space coverage for almost all instances in less than a second each.

5. SUMMARY AND CONCLUSIONS

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.

6. ACKNOWLEDGMENTS

This work is funded by the DARPA/MARCO Gigascale Silicon Research Center and an Agere Systems/SRC Research fellowship.

7. REFERENCES

[1] F. Aloul, M. Mneimneh, and K. Sakallah, "Backtrack Search Using ZBDDs," *in Int'l Workshop on Logic Synthesis*, 293-297, 2001.

[2] F. Aloul, J. Silva, and K. Sakallah, "An Experimental Study of Satisfiability Search Heuristics," *in the Proc. of Design, Automation and Test in Europe*, 2000.

[3] R. Bayardo Jr. and R. Schrag, "Using CSP look-back techniques to solve real world SAT instances," *in Proc. of the 14th National Conference on Artificial Intelligence*, 203-208, 1997.

[4] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking using SAT procedures instead of BDDs," *in Proc. of the Design Automation Conf*erence, 317-320, 1999.

[5] R. Bryant, "Graph-based algorithms for boolean function manipulation," *in Proc. of IEEE Trans. on Computers*, 35(8), 677-691, 1986.

[6] P. Chatalic and L. Simon, "Multi-Resolution on Compressed Sets of Clauses," *in Proc. of the Int'l Conference on Tools with Artificial Intelligence*, 2-10, 2000.

[7] J. Crawford, M. Ginsberg, E. Luks, and A. Roy, "Symmetry-Breaking Predicates for Search Problems," *in Knowledge Representation: Principles of Knowledge Representation and Reasoning,* 148-159, 1996.

[8] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem Proving," *in Comm. of the ACM*, 5(7), 394-397, 1962.

[9] DIMACS Challenge benchmarks in *ftp://Dimacs.rutgers.EDU/pub/challenge/sat/benchmarks/cnf*.

[10] D. Kokotov, I. Shlyakhter, "Progress Bar for SAT Solvers," unpublished manuscript, *http://sdg.lcs.mit.edu/satsolvers/progressbar.html,* 2000.

[11] M. Lobbing, O. Schroer, and I. Wegner, "The Theory of Zero-Suppressed BDDs and the Number of Knight's Tours," *in IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design, 1995.*

[12] J. P. Marques-Silva, "Algebraic Simplification Techniques for Propositional Satisfiability," in *Proc. of the Int'l Conference on Principles and Practice of Constraint Programming, 2000.*

[13] J. Marques-Silva and K. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *in IEEE Trans. on Computers*, 48(5), 506-521, 1999.

[14] S. Minato, "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems," *in Proc. of the Design Automation Conf.*, 1993.

[15] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *in Proc. of the Design Automation Conference*, 530-535, 2001.

[16] G. Nam, F. Aloul, K. Sakallah, and R. Rutenbar, "A Comparative Study of Two Boolean Formulations of FPGA Detailed Routing Constraints," *in the Proc. of the Int'l Symp. on Physical Design*, 2001.

[17] F. Somenzi, CUDD: CU Decision Diagram Package, University of Colorado at Boulder, *ftp://vlsi.colorado.edu/pub/*.

[18] M. Velev and R. Bryant, "Boolean Satisfiability with Transitivity Constraints," *in Proc. of the Conference on Computer-Aided Verification*, 86-98, 2000.

[19] H. Zhang, "SATO: An Efficient Propositional Prover," *in Int'l Conference on Automated Deduction*, 272-225, 1997.