Journal on Satisfiability, Boolean Modeling and Computation 5 (2008) 243–261

Research Note

Boosting SAT Solver Performance via a New Hybrid Approach *

Lei Fang leifang@vt.edu
Michael S. Hsiao mhsiao@vt.edu
Department of Electrical and Computer Engineering, Virginia Tech
Blacksburg, VA 24061, U.S.A.

Abstract

Due to the widespread demands for efficient SAT solvers in Electronic Design Automation applications, methods to boost the performance of the SAT solver are highly desired. We propose a Hybrid Solution to boost SAT solver performance in this paper, via an integration of local and DPLL-based search approaches. A local search is used to identify a subset of clauses from the original formula to be passed to a DPLL SAT solver incrementally until all the clauses have been passed. In addition, the solution obtained by the DPLL solver on the subset of clauses is fed back to the local search solver to jump over any locally optimal points. The proposed solution is highly portable to the existing SAT solvers. For satisfiable instances, up to an order of magnitude speedup was obtained via the proposed hybrid solver. For unsatisfiable instances, the speedup was smaller due to the overhead.

KEYWORDS: *satisfiability, DPLL, WalkSAT, hybrid*

Submitted January 2007; revised May 2008; published June 2008

1. Introduction

Boolean Satisfiability (SAT) has many applications, including computer-aided design, artificial intelligence, planning, etc. Various Electronic Design Automation (EDA) problems, such as equivalence checking, model checking, test pattern generation, placement and route, etc., can be formulated as SAT problems. Much research has been dedicated on developing highly scalable and efficient SAT solvers. Although a number of practical instances can be solved within reasonable computational resources by the state-of-the-art SAT solvers, due to the NP-Complete nature of SAT [8], many instances still remain extremely difficult.

A CNF-based SAT problem takes as input a propositional formula that is represented in Conjunctive Normal Form (CNF), in which a formula is a conjunction of clauses, each of which is a disjunction of literals. A literal is a variable in its positive or negative polarity. There are two broad, search-based approaches to modern search-based SAT solvers. One is based on a systematic search-tree and the other is based on stochastic local-search. The classical DPLL [9] algorithm and its derivatives like Chaff [18] and Minisat [11] are all search-tree based. In a search-tree-based algorithm, a variable is selected and assigned

* supported in part by NSF Grants 0305881 and 0417340

©2008 Delft University of Technology and the authors.

a value at each step. This assignment will be applied to the formula, after which new assignments of variables may be implied. If the current assignment to the variables causes a conflict, i.e., one or more clauses evaluate to false as all literals in those clauses evaluate to false, the solver will backtrack and make a different decision. In the process, conflict analysis can be performed to yield additional knowledge to aid future search. These steps are repeated until:

1. a solution is found, or
2. no satisfying solution exists, or
3. computational resources have been exhausted.

On the other hand, local-search solvers, such as GSAT [20] and WALKSAT [19], generate a complete assignment for all the variables at the beginning. Because every variable has been assigned, each clause is either satisfied or unsatisfied (all literals in the clause evaluate to false). A number of steps of local greedy search are applied next to try to minimize the number of unsatisfied clauses. For example, in WALKSAT, at each step a clause is selected from the current set of unsatisfied clauses, and the assignment to one of its literals is flipped. After a number of iterations, hopefully a solution can be obtained when the set of the unsatisfied clauses becomes empty. Nevertheless, it should be noted that it may be possible for the search to be trapped in a locally optimal point, where at least one unsatisfied clause still remains. The local optima here means an assignment in the solution space which can satisfy an equal or greater number of clauses than all its neighbors, but fails to satisfy all the clauses when the formula is actually satisfiable. To avoid being trapped indefinitely in local optima, a CUTOFF threshold is used to denote the maximum number of steps allowed. If the CUTOFF threshold is reached without obtaining a solution, a re-start in the solver may be invoked.

Generally, search-tree based algorithms are complete while local search algorithms are incomplete (Fang had published a complete local search algorithm [12]). Although local search algorithms can perform faster on a number of applications, they are hindered by their incompleteness.

Because both DPLL search and local search have their own strengths, there have been various attempts to combine them [15] [17]. In [17], a local search was used to help the DPLL solver select the next decision variable. Because the decision order of the DPLL SAT solver is directly modified by the local search part, the underlying decision order heuristics may potentially be degraded. This approach may not gain much performance as shown in [13]. In [15], WALKSAT was used to exploit the variable equivalences and dependencies at certain nodes of the DPLL tree.

In this paper, we propose a new hybrid framework to integrate local search and DPLL search for SAT. This integration offers completeness in the search and does not change the decision order heuristic in the underlying DPLL SAT solver. The non-deterministic local search attempts to find an assignment that can satisfy as many clauses as possible in the formula; any subset of clauses not satisfied is obtained and passed to a DPLL SAT solver through an incremental solver interface [7]. In return, the solution obtained for this subset of clauses by the DPLL solver is fed back to the local search solver to jump over

any local optimal points. In order to take advantage of both the local and DPLL search strategies, clause padding is proposed to make the integration both efficient and seamless. For satisfiable instances, up to an order of magnitude speedup was obtained over the state-of-the-art SAT solvers via the proposed hybrid solver. For unsatisfiable instances, smaller speedups were observed.

The remainder of the paper is organized as follows: The next section describes the preliminaries of the DPLL and the WALKSAT algorithms. The details behind our hybrid incremental SAT solver are presented at Section 3. Section 4.1 discusses the synergies between the DPLL and WALKSAT via some case studies. Section 4.2 presents the empirical evaluation of our proposed SAT solver. This paper is concluded in Section 5.

2. Preliminaries

We first provide a quick overview of the DPLL and the WALKSAT algorithms in this section. The DPLL algorithm was first published by Davis, Putnam, Logeman and Loveland [9] [10], and it is the basis for most modern SAT solvers. The pseudo-code of DPLL is listed in Figure 1.

As is shown in Figure 1, the DPLL algorithm works iteratively from line 4 to line 8. The decide() function in line 4 incorporates the decision order heuristic, which decides which unassigned variable should be assigned in this iteration. After a variable was assigned, Boolean Constraint Propagation (function bcp()) is called in line 6. In function bcp(), the BCP in [18] is implemented, where the current assignments are applied back to the formula to generate implications on the unassigned variables. This is the most time-consuming part in a given iteration of the DPLL algorithm because the formula is evaluated under the current variable assignments to check if any other variable assignments may be implied, or if a conflict has been encountered. If a conflict is met, function ResolveConflict() will be called in line 7. This function tries to resolve the conflict by backtracking, shown between line 12 to 18. An UNSAT result will be returned if a conflict is not resolvable (ResolveConflict() returns false). Otherwise the algorithm goes back to line 6 to invoke function bcp() again. When there is neither future implication nor conflict available, the algorithm will be looped back to line 4 to select the next decision variable. One can see that if the formula is satisfiable, an assignment, A, to the variables will be generated eventually.

WALKSAT [19] was proposed in 1994 as an improvement of the previous GSAT [20]. First, several definitions will be given for the WALKSAT algorithm, which is shown in Figure 2. A clause is said to be *broken* if all the literals in this clause evaluate to false under the current variable assignment. The *broken-count* of a variable is the number of clauses that will be broken if the value of this variable is flipped. The broken-count of a variable is used to evaluate the gain from flipping the given variable. When a broken clause becomes non-broken by flipping the value of a variable, this clause is referred to as having been *healed* by the flipping. For example, given the formula $(\bar{a}+b)(b+c)(a+\bar{c})$ and the assignment $\{a=1, b=0, c=0\}$, clauses $(\bar{a}+b)$ and $(b+c)$ are *broken* by this assignment. If the value of variable a is flipped from 1 to 0, clause $(\bar{a}+b)$ will be *healed* by this flip and only clause $(b+c)$ will be left as *broken*. After the flip, the broken-counts of the variables $\{a, b, c\}$ are $\{1, 0, 1\}$.

```
1  def DPLL
2  begin
3      while (true)
4          if (!decide()) /*if no unassigned vars*/
5              return SAT
6          while (!bcp())
7              if (!ResolveConflict())
8                  return UNSAT
9  end
10 def ResolveConflict()
11 begin
12     d = most recent decision not  tried bothways
13     if (d == NULL) // no such d was found
14         return false;
15     flip the value of d;
16     mark d as tried both ways;
17     undo any invalidated implications;
18     return true;
19 end
```

Figure 1. DPLL Algorithm

```
1  def WALKSAT
2  begin
3      A=randomly generate truth assignment
4      for i=1 to CUTOFF
5          if (A satisfies the formula)
6              return A /*SAT*/
7          C=choose a broken clause
8          if (rand() %100 < p) /*with prob. p*/
9              v= var with smallest broken-count in C
10         else /*with probability 1-p*/
11             v= randomly select a variable in C
12         Flip(v) /*reverse value of v*/
13         UpdateAssignment(A) /*A is updated with the value of v reversed*/
14     return FAIL /* can't determine */
15 end
```

Figure 2. WALKSAT Algorithm

The major body of the WALKSAT algorithm is the loop shown in Figure 2 between lines 4 and 14. Note that WALKSAT will try a preset number of steps before it claims that it has failed to find a solution. The preset number is stored in variable CUTOFF. First a complete assignment of all the variables is generated randomly as the start point of the following flips. In line 7, a variable v is chosen from a broken clause C; the value of v is flipped in an attempt to reverse the conflict currently caused in C. Normally v is selected among several variable candidates by comparing the potential gains of the eligible candidates. When the value of a variable v is flipped, some broken clauses may be healed while some previously satisfied clauses may be broken. In the basic WALKSAT, a variable with the smallest broken-count will be chosen to be flipped, as shown in line 9. Besides, certain randomness is introduced in lines 10 and 11. In line 12 and 13, the assignment

of v is flipped and its associated broken-counts are updated. To efficiently update the broken-count, a two-literal watching scheme was proposed in [14].

3. Our Approach

In order to combine the powers of both local search and DPLL-based search, previous approaches mainly tried to embed the local search result into a DPLL solver to guide the decision order. In such approaches, the local search is invoked at each DPLL decision step to supply the information for the next decision. On the contrary, in our approach, the local search portion is used to identify a subset of clauses, which is passed to a DPLL-based incremental SAT solver. Furthermore, the solution obtained by the incremental DPLL solver on the subset of clauses is fed back to the local search solver to jump over the locally optimal points encountered in the previous iteration in order to continue the search.

We call our solution HBISAT (HyBrid Incremental SAT Solver). It should be noted that HBISAT does not necessarily rely on a specific SAT solver. HBISAT actually is a universal solution to boost existing SAT solver performance.

We use the WALKSAT v43 [1] algorithm as the base local search part. In the rest of the paper, the local search solver will be referred to as WALKSAT. The complete HBISAT algorithm is shown in Figure 3.

In Figure 3, the DPLL solver and WALKSAT are invoked interactively in each iteration between lines 8 and 22. In each iteration, WALKSAT first performs a local search in an attempt to find a satisfying assignment for the entire Boolean formula. If it is successful, then HBISAT will verify the solution and return it. In Figure 3, the aforementioned WALKSAT search is presented between lines 12 and 14. If WALKSAT cannot find a solution within its CUTOFF value, it will collect the subset of broken (unsatisfied) clauses (B_i) under the current assignment and add them to the clause database of the DPLL solver, shown at lines 26 through 30 in function CallDPLL(). Note that the initial clause database, C_0, of the DPLL solver is empty, and C_{i-1} (for $i > 1$) denotes the subset of clauses that has already been added into the DPLL solver before the current i^{th} iteration. If the DPLL solver can prove $C_i = B_i \bigcup C_{i-1}$ is UNSAT in any iteration i, then the original formula is guaranteed to be UNSAT due to the reason that C_i is always a subset of the original formula; this allows for early termination of the SAT search.The detailed proof is provided in Theorem 1. On the other hand, if an assignment is obtained by the DPLL solver for C_i, the variable assignment will be passed back to WALKSAT as the starting assignment for the next iteration, at line 18 in Figure 3. Note that since C_i may only contain a portion of the variables, the assignment returned by the DPLL solver could be incomplete; those variables not present in this assignment retain their values from the previous iteration. We can see that the set of clauses added into the clause database of the DPLL solver grows gradually with the increasing number of iterations. This is the underlying essence of a typical incremental SAT solver, and it is also key to our hybrid incremental framework. To avoid adding the same broken clause to the DPLL solver repeatedly, in our implementation, each clause in the original formula processed by WALKSAT carries a flag to indicate whether it has been added to the clause database of the DPLL solver or not. Another advantage of employing an incremental SAT solver is that conflict clauses obtained by the DPLL solver can be carried from one iteration to the next [21].

```
1  def HBISAT_Solver(F/*input formula*/)
2  begin
3      C=∅ /*Set of broken clauses*/
4      A=∅ /*Assignment*/
5      i=0 /*iteration counter*/
6      CUTOFF=MAX_LOCAL_SEARCH_STEP
7      while (true)
8          if (A==∅)
9              RandomInitalWalkSAT()
10         else
11             InitialWalkSAT(A)
12         Status=WALKSAT_Solver(CUTOFF)
13         if (Status==SAT)
14              return SAT
15         else if (Status==UNKNOWN)
16             Result=CallDPLL(i)
17             if (Result==SAT)
18                 A=GetAssignmentFromDPLL()
19             else /*UNSAT or out of memory*/
20                  return Result
21     i++ /*count iteration*/
22 end
23
24 def CallDPLL(i)
25 begin
26     B=GetBrokenclause()
27     /*guarantee one more clause will be added*/
28     B = B ⋃ RandomPickOneNewClause()
29     C = B ⋃ C
30     AddClauseToDPLLSolver(C)
31     Status=DPLL_Solver()
32     return Status
33 end
```

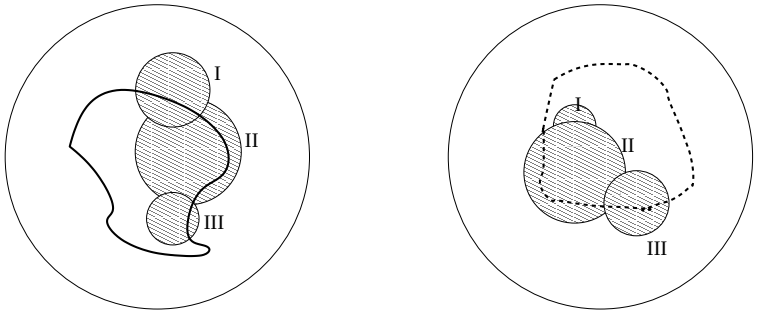
Figure 3. HBISAT Algorithm

We will now prove that HBISAT is a complete algorithm, in which it finds a solution if one exists; otherwise it reports no variable assignment can satisfy the formula.

Theorem 1. *HBISAT is complete.*

Proof. In the HBISAT Algorithm shown in Figure 3, whenever B_i from iteration i is empty, which means WALKSAT already found a solution that generates zero broken clauses, then a satisfying solution has already been found for the formula. Otherwise, with a non-empty B_i at iteration i, we know that B_i contains at least one clause from the original formula that has not been included in the clause database of the DPLL solver previously. This is due to at each iteration $B_i = B_i \bigcup$ RandomPickOneNewClause(), at line 28 in Figure 3. In function RandomPickOneNewClause(), a clause that has not been added to the DPLL solver is randomly picked to guarantee that B_i contains at least one "new"clause. Let the size of the original formula be n, and the clause database at iteration i of the DPLL solver be DB_i. Then, by gradually adding each B_i to the clause database of the DPLL solver, the size of the DPLL database increments by at least 1 with each iteration. In other words, $|DB_{i+1}| \geq |DB_i| + 1$. As the algorithm proceeds, if the DPLL solver returns UNSAT, then an UNSAT core has been identified and the original formula is indeed unsatisfiable and

HBISAT is complete. Otherwise, $|DB_i|$ will eventually equal to n, where DB_i contains the entire original formula. Because the DPLL solver is complete, we can conclude that HBISAT is also complete. □



(a) UNSAT formula (b) SAT formula

Figure 4. Operation of HBISAT on UNSAT and SAT formulas

Figure 4 illustrates how HBISAT works on a given formula. Figure 4(a) shows the scenario that the formula is unsatisfiable and Figure 4(b) is for the scenario when the formula is satisfiable. In both parts (a) and (b) of the figure, the outermost circle represents all the clauses of the formula to be solved. The inner, shaded circles represent the sets of broken clauses. The annotations I, II and III on the inner circles denote that they are three different broken clause sets generated from three different iterations. Note that shapes I, II and III are not necessarily overlapped because at each iteration the set of broken clauses C_i could be totally different. They are overlapped only when their corresponding broken clause sets are not disjoint.

If the original formula is unsatisfiable, there exists at least one unsatisfiable core [23]. Without loss of generality, let us assume there is only one unsatisfiable core, and this core is represented by the region enclosed by the thicker line in part (a) of the figure. Each set of broken clauses must cover a portion of the unsatisfiable core (as will be proved in Theorem 2). In each iteration of HBISAT, a portion of the unsatisfiable core will be identified. With an increasing number of iterations, the entire unsatisfiable core will be extracted from the formula which leads to an early termination.

Theorem 2. *If a formula f is unsatisfiable, every broken clause set returned by WALKSAT contains at least one clause that belongs to an unsatisfiable core.*

Proof. We prove this by contradiction. Given that f is unsatisfiable, there always exists at least one unsatisfiable core $UC \subseteq F$ (F: set of clauses of f). Let a broken clause set, B, returned by WALKSAT not contain any clause belonging to UC, i.e., $B \bigcap UC = \emptyset$. This means that WALKSAT must have obtained an assignment that satisfies all clauses outside of B. However, because $B \bigcap UC = \emptyset$, the clauses outside B contain the complete UC. This means that all clauses within UC must have been satisfied by the obtained assignment, indicating that UC is satisfiable: a contradiction. □

From Theorem 2 we know that at each iteration, at least some portion of an unsatisfiable core will to be added to the clause database of the DPLL solver, if the original formula is unsatisfiable. Stated differently, HBISAT can filter the hard spots like the unsatisfiable core or hard-to-simultaneously-satisfy sets of the formula. Due to the nature of the DPLL-based incremental SAT solver, these hard spots will be solved incrementally and the conflict clauses learned through them have tremendous potential to help finally solve the formula.

On the other hand, if the formula is satisfiable, for hard satisfiable instances usually there will be one or more sets of clauses called hard-to-simultaneously-satisfy sets. Part (b) of Figure 4 shows such a scenario where the region enclosed by the dashed line represents a hard-to-simultaneously-satisfy set. Because WALKSAT attempts to find an assignment via a local search, we can assume that the assignment generated by WALKSAT is more likely to satisfy those easy clauses outside the hard-to-simultaneously-satisfy region first. With this assumption, the hard-to-simultaneously-satisfy sets will be identified by HBISAT gradually in a similar way as the unsatisfiable core. It is important to find and solve these hard-to-simultaneously-satisfy sets earlier because they are the major obstacles in solving the hard satisfiable formulas.

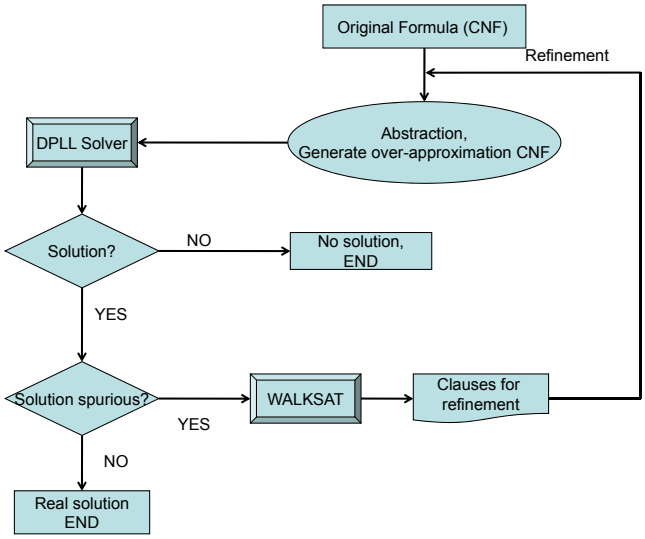


Figure 5. Abstraction and refinement in HBISAT

To better understand HBISAT, it can be explained as an abstraction and refinement scheme, shown in Figure 5 . Abstraction-refinement is a widely used technique in formal verification [6]. Because abstraction helps to reduce the complexities and refinement guarantees the correctness of the abstract model, an abstraction-refinement scheme usually can significantly boost the performance in many applications. Due to the fact that the set of clauses that the DPLL solver works on is a subset of the original formula, this set of clauses can be viewed as an abstraction of the original formula. We call this an abstraction model of the original formula with less constraints. Thus the partial assignment returned from the

DPLL solver actually can be regarded as an over-approximation of the true solution, since it satisfies an abstraction of the original formula. If we look at the solution spaces, one can see that the partial assignment represents a sub-space where true solutions might reside in. It should be noted that there is no guarantee that a true solution will be in the sub-space defined by the partial assignment. It is possible that there does not exist any solution in this sub-solution-space to satisfy the original formula. In other words, each solution in the sub-space can only satisfy the abstraction which includes less constraints compared with the original formula. Under such a circumstance this over-approximation solution is called *spurious* which indicates that the current abstract model is not accurate enough to lead to a true solution. Then the refinement will be invoked to calibrate a better abstraction. In HBISAT the WALKSAT takes over the over-approximation solution to verify whether it is spurious or not, along with a local greedy search to find a real solution. If a real solution can't be found by WALKSAT, a set of broken clauses B will be provided as the refinement, where B will be added to the clause database of the DPLL solver to make the abstraction more precise.

HBISAT is a flexible SAT framework. Instead of the specific local search and DPLL solvers we used in this paper, other such engines can be plugged in. The interaction between our local-search solver and DPLL-search solver is simple, which is supported by most modern SAT solvers. The requirements for the DPLL-based solver are that it should support an incremental interface and be able to return its search results. For the local search solver, it must be able to supply the set of broken clauses. We believe the flexibility of HBISAT offers a significant value, because it holds potential for future explorations.

3.1 Clause Padding

In a given iteration, the number of broken clauses may be small, which implies that a large number of iterations may be needed before all clauses in the original formula are added to the clause database for the DPLL solver. Therefore, in addition to the broken clauses from WALKSAT, some other related clauses may also be inserted into the DPLL solver. This feature is called *clause padding*. By adding more clauses to the DPLL solver at each iteration, there are two potential benefits. First, clause padding may help the DPLL solver to find the unsatisfiable core earlier since adding more clauses further constrains the problem, and second, it can speed up the incremental SAT solver process. The padded clauses are chosen mainly based on their correlations to the broken clauses.

In HBISAT two categories of clauses are padded in the clause padding procedure:

1. Based on the assumption that the flip frequency of a variable (how many times a variable flipped) usually indicates its importance of solving the formula, the clauses which contains the most frequently flipped variable are padded to the clause database of the DPLL solver.

2. We put all the literals of broken clauses in the current iteration into an array R. Then any clause with two or more of its literals having opposite polarities with the corresponding literals in R will be padded into the clause database. The intuition here is that we want to pad those clauses that are highly correlated with the broken clause set, with the hope that they will help to constrain the SAT solver.

3.2 Enhance WALKSAT with Conflict Clauses from DPLL

The learned conflict-induced clauses during the DPLL search process could be very powerful to block unnecessary search spaces. Because the DPLL solver is called through an incremental interface, naturally it will take advantage of those conflict-induced clauses inherited from the previous iterations. However, its counterpart (WALKSAT) cannot directly benefit from these conflict clauses. Since WALKSAT works without history information in each iteration, no helpful information will be carried over to the next iteration. To overcome this limitation, the conflict clauses are introduced back to the original formula in WALKSAT, which means that WALKSAT will be operated on a set of clauses constituted by the original formula plus the conflict clauses generated by the DPLL solver. Note that this new set of clauses constitutes an expanded formula. The expanded formula may vary from an iteration to the next. Given the fact that the conflict clauses represent some hard-to-discover implications from the original formula, they could help WALKSAT to reach a minimum faster. We will illustrate this with the following example.

Considering the CNF formula $(a+c)(b+\bar{c})(\bar{a}+\bar{b}+c)(a+\bar{b}+\bar{c})(\bar{a}+b+c)(\bar{a}+b+\bar{c})$, with an initial assignment of $abc = 100$, the broken-count is $\{1, 1, 2\}$. If we pick the broken clause $(\bar{a} + b + c)$, it is not easy for WALKSAT to make a clear decision because variables a and b tie on the broken-count. Randomly flipping a will be a step in a wrong direction because (a) can only be 1 in order to satisfy the formula. Suppose an additional conflict clauses $(a + b)$ returned by the DPLL solver was added to this formula, then the broken-counts would have been updated as $\{2, 1, 2\}$. Based on the broken-counts $\{2, 1, 2\}$, WALKSAT flips variable b and that leads to a new assignment $\{1, 1, 0\}$ with broken-counts $\{1, 1, 0\}$. Now in the broken clause $(\bar{a}+\bar{b}+c)$, variable c will be chosen to be flipped due to its lowest penalty on broken-count, thus a solution $abc = 111$ will be obtained.

When applying WALKSAT to the expanded formula, the effectiveness of conflict clauses could be overshadowed by the huge number of clauses in the original formula. In order to leverage the importance of the conflict clauses, they are assigned a bigger weight in broken-count counting, which is 2 in our implementation.

4. Experimental Results

4.1 Study of Synergies Between WALKSAT and DPLL Solvers

In this subsection, we will explore the interactions between the WALKSAT and DPLL solvers through some benchmarks. We will illustrate the discussion via a number of figures; in each figure, the X-axis denotes the iteration index and the Y-axis denotes the percentage of clauses. In Figure 6, results for benchmark c10_s.cnf from [16] are shown. In this case, an early termination case is evident (HBISAT stopped before 100% clauses in the original formula were added to the clause database of the DPLL solver). The top curve with stars represents the number of clauses that are added to the DPLL solver at each iteration step. We call it the "ADD curve." The bottom curve shows the number of broken clauses returned by WALKSAT, which is called the "BRK curve." It can be observed that when 90% of the clauses have been added, the DPLL solver can conclude that the instance is UNSAT. Although it does not necessarily mean that the current clauses in the DPLL solver

form a minimal UNSAT core, the early termination provides the potential to reduce the computational effort for solving the complete formula.

Figure 7 presents a case where WALKSAT becomes more effective when guided by the solutions returned from the DPLL solver. The benchmark used here is pipe-64-4-bug01.cnf in Table 3 [22]. One can observe that the BRK curve generally drops with the increasing number of iterations. This is because the partial assignments supplied by the DPLL solver give a better guide to the WALKSAT. Another interesting phenomenon is that the ADD curve grows much faster at certain points. This is due to the clause padding mechanism. It is possible that a small group of broken clauses returned by WALKSAT can induce a larger group of clauses to be padded, where the padded clauses help to increase the chance of conflicts in the DPLL solver. After 14 iterations, a solution is found by WALKSAT for this satisfiable instance. In contrast, the pure WALKSAT cannot obtain the solution for this instance after 200 iterations with the same CUTOFF value in each iteration.

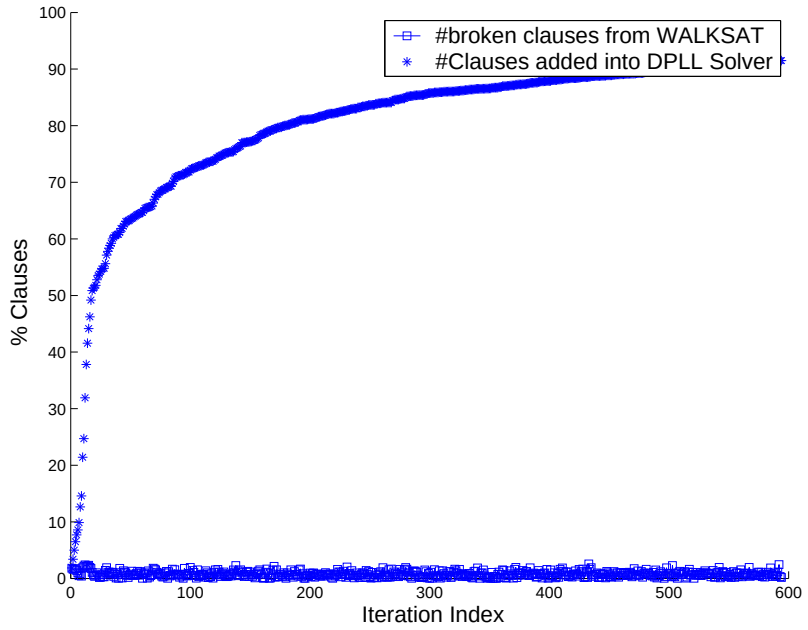


Figure 6. Early termination of an UNSAT instance, c10_s.cnf

If early termination is not achievable and a solution cannot be found by WALKSAT, the entire instance will eventually be added to the DPLL solver. Figure 8 illustrates how the conflict clauses help to finally solve the instance with the benchmark Mat25.shuffled.cnf from the SAT competition 2002 [2]. Since the discrepancy between the two curves in the figure represents the number of conflict learned clauses, we can see that more conflict learned clauses emerge when more clauses are added to the DPLL solver. This is easy to understand because a more strongly correlated clause set has a higher probability for conflict learning. After 90 iterations, by measuring the discrepancy between the two curves, one can see that more than 10% of the clauses in the DPLL solver are conflict clauses. At this point, when HIBSAT adds all the clauses from the original instance to the DPLL solver it takes 4.48

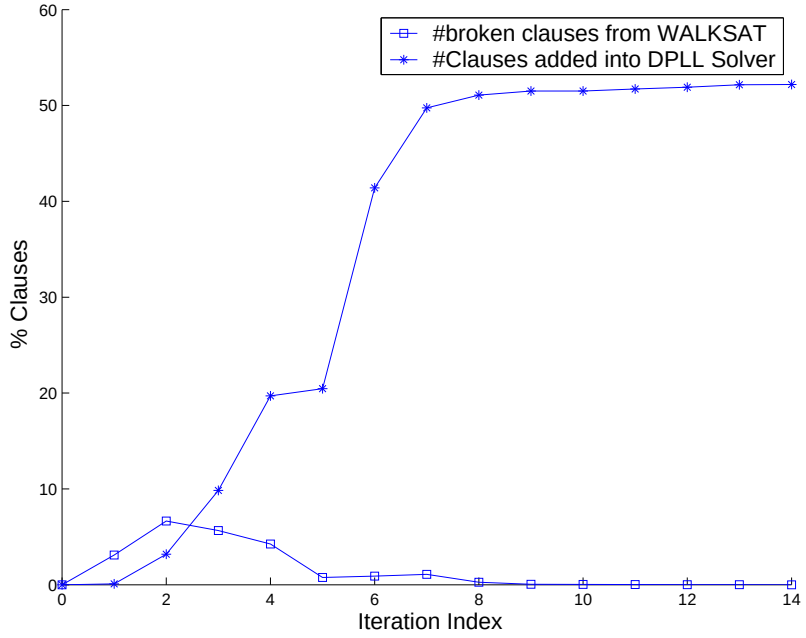


Figure 7. Solution found at an early stage by WALKSAT after 15 iterations in pipe-64-4-bug01.cnf, with less than 55% clauses in the original formula added to the DPLL solver

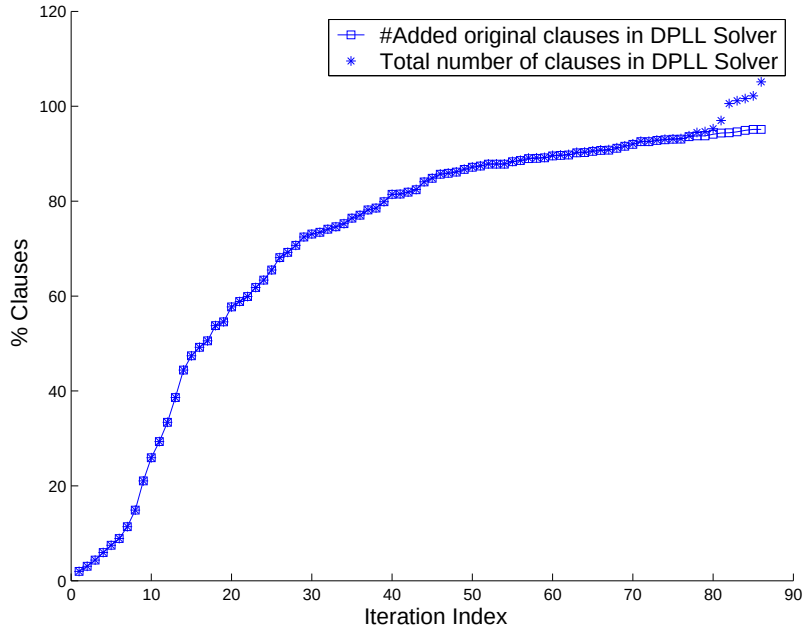


Figure 8. Example of gradually learning in Mat25.shuffled.cnf

seconds to prove UNSAT with a total running time of only 6.26 seconds. On the other hand, solving the original instance directly by the DPLL solver requires 15.07 seconds. The gradually learned clauses can significantly boost the DPLL solver's performance.

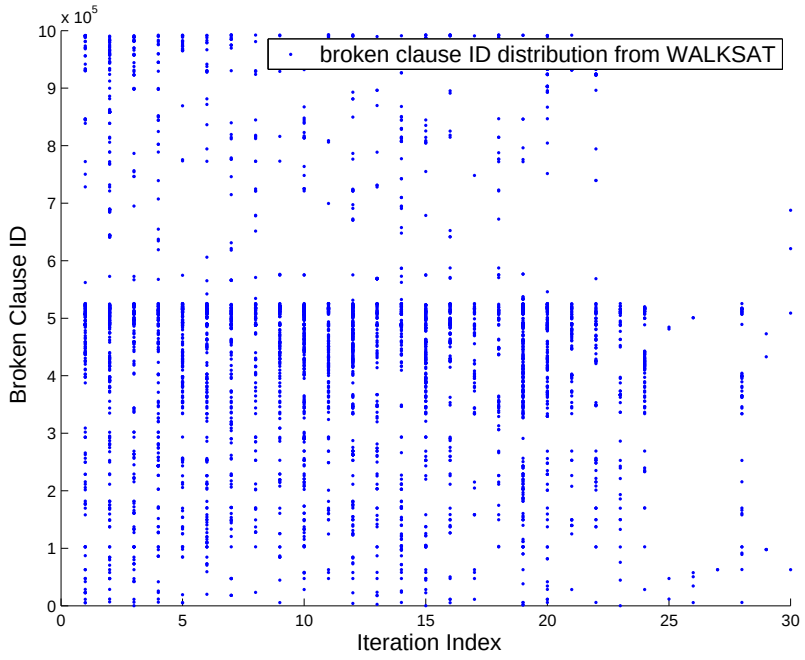


Figure 9. Broken clauses distribution for pipe-64-4-bug03.cnf

Figure 9 shows an example about the distribution of broken clauses in each iteration for benchmark pipe-64-4-bug03.cnf in Table 3 [22]. The X-axis is again the iteration indices but the Y-axis now denotes the clause ID. If a clause with a clause ID i is returned by WALKSAT as a broken clause at iteration j, position (i, j) in the figure will be dotted. While the size of the broken clauses does not monotonically decrease during HBISAT's solving process, it can be observed that at higher iteration numbers, the size of the broken clauses does begin to shrink.

4.2 More Results

The proposed HBISAT was implemented in C++ under the 64-bit Linux operating system, and experiments were conducted on a Intel Xeon 3.0-GHz workstation with 2 GB of RAM. To demonstrate the portability of our algorithm, HBISAT was built on top of two popular SAT solvers, ZChaff version 2004.11.15 Simplified [3] and Minisat 1.14 [4]. They are called HBIz and HBIm in the experiments. Because we are interested in improving the SAT performance of EDA applications, all of the benchmarks chosen are hard publicly available EDA instances. Three categories of benchmarks were used in our experiments. The first category is the IBM Formal Verification Benchmarks Library [5]. Several groups of instances were chosen from the library and each group contained six instances. Note that each group corresponds to a bounded model checking (BMC) application, where instances in a group

represent different lengths of time-frame expansion. The second category comes from a parameterized benchmark suite of Hard-Pipelined-Machine verification problem [16], which includes twelve instances. The third category contains fifteen instances which are generated in the formal verification of buggy variants of an out-of-order superscalar processor from CMU [22]. In order to evaluate the performance of HBISAT, results for both DPLL solvers (ZChaff and Minisat) and their corresponding HBISATs were reported. We believe similar results can be obtained if other different SAT solvers were used in place of ZChaff or Minisat, as the framework for HBISAT requires only that the underlying DPLL solver includes an incremental solver interface. Finally, because completeness is needed to handle UNSAT instances, pure local-search based SAT solvers were not compared. In fact, for most of the satisfiable instances in the experiments, the pure WALKSAT could not complete.

In our implementation, in order to avoid unnecessary iterations, whenever one of the following three conditions is met, *all* the remaining clauses will be added into the DPLL solver. The first condition is when less than one percent clauses is left that have not been added to the DPLL solver. The second condition is if less than fifty clauses are left, for cases where fifty clauses are greater than one percent of all clauses in the formula. The third condition is if the number of learned clauses is more than twenty percent of the number of original clauses. We also limit the number of iterations to 300.

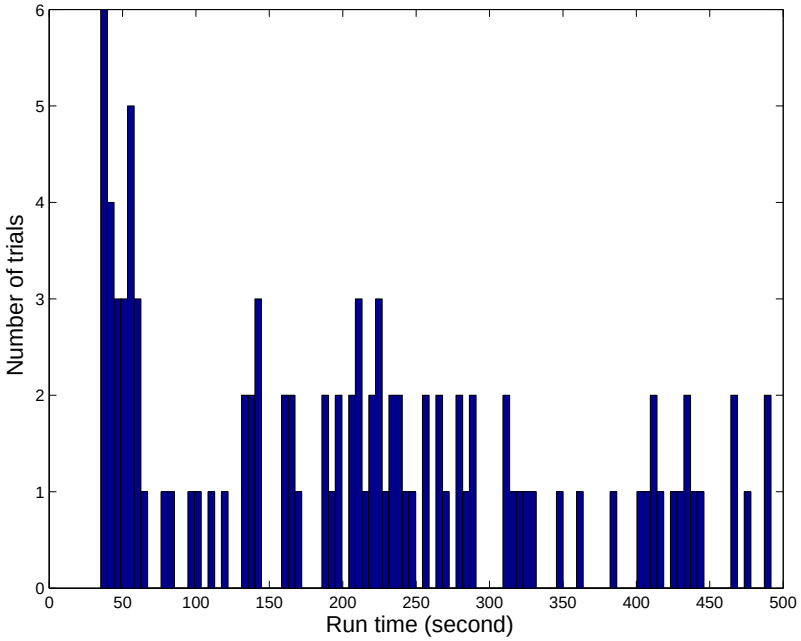


Figure 10. Histogram of run time distribution for 01.k50

As WALKSAT is embedded within HBISAT, there is a certain level of randomness in its searching process. To investigate the robustness of HBISAT, a case study was set up to show the run time distribution. In this experiment, HBIz was called 100 times on the benchmark. The benchmark we will use to illustrate this is 01.k50 from Table 1 [5]. In Figure 10, the histogram of the 100 trials was shown with each bin size of 10, where the

X-axis is the run time and Y-axis shows the number of trials that fall into this time period. One can see that although the run time varies, the trend is clear that for most runs, the run-time is more likely to be small. In other words, the probability of larger run time decreases. One key point to note is that except for the very first iteration, in all the rest of the iterations the embedded WALKSAT always starts from the partial assignment returned by the DPLL solver. This can help reduce the randomness in the algorithm. For all the experimental results in the following are the average value of five runs on each benchmark.

Table 1. Category I Benchmarks

Instance	#Var/#Cls	SAT?	ZChaff [s]	HBIz [s]	Speedup	#Iter
01.k40	29249/124460	SAT	99.26	**61.73**	1.61×	41
01.k50	36339/154810	SAT	OUT	**177.89**	INF	29
01.k60	43429/185160	SAT	OUT	**490.74**	INF	17
03.k40	46225/198298	SAT	31.45	57.56	0.55×	14
03.k50	58595/251378	SAT	258.28	**195.81**	1.32×	11
03.k60	70965/304458	SAT	434.55	**250.99**	1.73×	11
06.k40	49126/213666	SAT	124.31	**115.54**	1.08×	193
06.k50	61776/268886	SAT	1921.92	**247.1**	7.78×	15
06.k60	74426/324106	SAT	1709.39	**1605.57**	1.06×	10
07.k40	13151/35904	UNSAT	6.04	20.59	0.29×	300
07.k50	15221/40774	UNSAT	5.94	19.19	0.31×	300
07.k60	17291/45644	UNSAT	6.06	20.58	0.29×	300
Total			8915.4	**3270.57**	2.73×	
Instance	#Var/#Cls	SAT?	Minisat [s]	HBIm [s]	Speedup	#Iter
01.k70	50519/215510	SAT	53.89	**53.01**	1.02×	34
01.k80	57609/245860	SAT	174.8	**132.74**	1.32×	32
01.k90	64699/276210	SAT	330.41	**60.03**	5.48×	26
03.k70	83335/357538	SAT	45.84	76.39	0.60×	42
03.k80	95705/410618	SAT	81.71	**80.19**	1.01×	36
03.k90	108075/463698	SAT	267.64	**197.57**	1.36×	30
06.k70	87076/379326	SAT	60.22	109.09	0.55×	36
06.k80	99726/434546	SAT	201.19	**135.64**	1.48×	30
06.k90	112376/489766	SAT	296.07	**189.53**	1.56×	29
07.k70	19361/50514	UNSAT	9.86	15.33	0.64×	300
07.k80	21431/55384	UNSAT	1065.7	**17.45**	60.76×	300
07.k90	23501/60254	UNSAT	5.97	16.21	0.37×	300
Total			2593.7	**1077.86**	2.41×	

We first report the results for Category I benchmarks, shown in Table 1. The upper portion represents the comparison between ZChaff and HBIz. The bottom portion compares Minisat and HBIm. A total of four groups (24 instances) is listed in the first column. Next, the number of variables and clauses for each instance are reported, followed by the satisfiability of each instance. The forth and fifth columns correspond to the run times of the DPLL solvers and its HBISAT. Whenever HBISAT outperformed the DPLL solver, the run times are highlighted in bold. "OUT" indicates that the SAT solver aborts after a preset limit is met, which is 3,000 seconds in our experiments. The sixth column exhibits the speedup of HBISAT, which was calculated by computing the ratio between the run time the DPLL solver and HBISAT. Note that INF means at least one of the solvers time out thus no run time ratio can be computed. Finally, the last column reports the number of iterations the HBISAT used.

It can be observed that among all the twenty-four Category-I instances, ZChaff aborted on two of them while HBIz could solve every instance within 500 seconds, except 06.k60.

Table 2. Category II UNSAT Benchmarks

Instance	#Var/#Cls	ZChaff [s]	HBIz [s]	Speedup	#Iter	Minisat [s]	HBIm [s]	Speedup	#Iter
c7b	26058/77128	147.4	149.3	0.99×	300	55.54	**54.27**	1.02×	300
c8n	53697/159595	386.93	**363.89**	1.06×	300	146.18	169.25	0.86×	300
c9b	36757/109045	407.53	**381.7**	1.07×	300	263.54	**226.02**	1.17×	300
f9n	185149/552412	2011	**1604.91**	1.31×	300	OUT	OUT	INF	300
g9n	54631/161950	251.33	**222.53**	1.13×	300	61.84	84.12	0.74×	300
g9b	59110/175387	208.65	233.03	0.90×	300	41.49	106.8	0.39×	300
g9idw	125885/371998	250.55	263.39	0.95×	300	70.64	136.11	0.52×	300
g9nidw	170918/506584	1552.48	**1098.25**	1.41×	300	942.97	**691.3**	1.36×	300
c10	17121/50803	9.87	16.25	0.61×	300	12.15	24.44	0.50×	300
c10b	43517/129265	669.88	**529.1**	1.27×	300	589.1	**567.78**	1.04×	300
c10bi	147116/437224	OUT	OUT	INF	300	OUT	OUT	INF	300
c10bid	291912/828039	OUT	OUT	INF	300	OUT	OUT	INF	300
Total		5476	**4549.4**	1.20×		2183.45	**2060.09**	1.04×	

HBIz outperformed ZChaff in 8 of the 12 instances. For the other four instances, they were all easy instances. For such easier instances, the overhead of HBISAT became a burden. On the other hand, with larger BMC instances (due to deeper circuit unrolling), the computational costs of ZChaff were increased dramatically while the run times of HBIz increased relatively linearly. Similarly HBIm outperformed Minisat in 9 of 12 instances. For the UNSAT instance 07.k80, HBIm only took 17.45 second while Minisat's running time is 60 times that. One can see that the number of iterations varies from 11 to 300. For those UNSAT cases HBISAT could not finish within 299 iterations, at the 300^{th} iteration, all clauses are added into DPLL Solver, since this is the last iteration allowed. The rest of the benchmarks take less iterations due to reason that after a few iterations HBISAT already accumulated enough learned clauses thus to trigger adding of all clauses, as mentioned before.

Next, experiment II was set up to evaluate the unsatisfiable instances, with the results reported in Table 2. The first six columns of Table 2 are similar as in Table 1. The run time of Minisat, HBIm, the corresponding speedup and the iteration number of HBIm were reported in the last four columns. The performance of HBISAT was comparable to the DPLL solver for most instances. Minisat and HBIm aborted on exactly three instances and the average run times for the other instances were nearly equal. Meanwhile, Zchaff and HBIz aborted on two instances. Generally HBIz gives better performance on harder instances thus leads to a total 900 seconds run time reduction. The reason that the performance gain on unsatisfied instances sometimes was not as significant can be explained by the nature of incremental SAT solvers. For hard UNSAT instances, it may not be easy to obtain a small UNSAT core (or a superset of the UNSAT core) from the original formula. Subsequently, the incremental clause databases may all be satisfiable, and we may need to wait until nearly all the original clauses have been added before concluding that the formula is UNSAT. Furthermore, during the incremental steps, the subset of clauses currently in the database of the DPLL solver may constitute a hard satisfiable instance, and this hard satisfiable instance may consume significant computational resources. On the other hand, the non-incremental DPLL solver searches directly on the entire formula, where such hard intermediate steps are implicitly avoided.

Finally, the results for Category III benchmarks are shown in Table 3. For all the SAT instances, HBISAT exhibits a very significant performance gain, where HBISAT achieved nearly an order of magnitude reduction in run times. For instance, in benchmark bug07, where Minisat failed in 3000 seconds and ZChaff took 997.67 seconds, HBIz took 28.41 seconds with 41 iterations while HBIm needed only 15.01 seconds in 66 iterations. The small number of iterations here is due to the fact that a solution can be found by the local search at an early stage. The power of combining local search and DPLL search is evident via experiments I, II and III. In terms of UNSAT instances in both experiments II and III, HBISAT performances comparably with its DPLL counterpart with acceptable overhead on some instances. The experimental results between HBIz and HBIm are consistently matched, which demonstrate the flexibility and effectiveness of our proposed hybrid framework.

Table 3. Category III Benchmarks

Instances	#Var/#Cls	SAT?	ZChaff [s]	HBIz [s]	Speedup	#Iter	Minisat [s]	HBIm [s]	Speedup	#Iter
bug01.cnf	35853/1021170	SAT	40.83	**8.17**	5.00×	21	1.55	4.29	0.36×	3
bug02.cnf	35853/1021171	SAT	OUT	**14.81**	INF	38	OUT	OUT	INF	300
bug03.cnf	35947/992674	SAT	4.8	11.19	0.40×	30	0.97	4.28	0.23×	10
bug04.cnf	35854/1012315	SAT	39.07	**11.16**	3.5×	24	4.39	7.84	0.56×	9
bug05.cnf	35853/1022271	SAT	555.54	**40.86**	13.60×	97	OUT	OUT	INF	300
bug06.cnf	35853/1022271	SAT	282.08	**6.67**	42.30×	27	OUT	**9.4**	INF	13
bug07.cnf	35853/1022271	SAT	997.67	**28.41**	35.11×	41	OUT	**15.01**	INF	66
bug08.cnf	35622/1003074	SAT	38.5	**6.44**	5.98×	13	OUT	**4.63**	INF	6
bug09.cnf	35726/1011764	SAT	0.4	10.09	0.04×	26	180.03	**5.29**	38.88×	6
bug10.cnf	35839/1012135	SAT	0.62	12.18	0.05×	20	1242.21	**3.6**	345.06×	3
bug11.cnf	35853/1012271	SAT	1148.21	**93.65**	12.26×	148	2.6	10.37	0.25×	42
Total			3107.73	**243.63**	12.6×		1431.75	**64.71**	22.13×	
5pipe.cnf	9471/195452	UNSAT	10.99	30.78	0.36×	300	78.77	185.66	0.42×	300
5pipe-5-ooo.cnf	10113/240892	UNSAT	34.30	52.75	0.65×	300	OUT	OUT	INF	300
6pipe.cnf	15800/394739	UNSAT	86.98	**66.52**	1.30×	300	OUT	OUT	INF	300
6pipe-6-ooo.cnf	17064/545612	UNSAT	159.35	276.63	0.58×	300	252.53	**205.35**	1.23×	300
7pipe.cnf	23910/751118	UNSAT	275.58	284.23	0.97×	300	OUT	OUT	INF	300
7pipe-7-ooo.cnf	24415/711050	UNSAT	2768.58	**2425.08**	1.14×	300	OUT	OUT	INF	300
Total			3335.78	**2926.7**	1.14×		**349.74**	381.95	0.92×	

5. Conclusions

In this paper, a new Hybrid SAT solver framework (HBISAT) was presented that combines the power of local search and modern DPLL-based search with conflict-driven learning. In HBISAT, the local search interacts with a DPLL SAT solver incrementally. In effect, the local search helps to identify incremental sets of clauses that are hard, and these clauses are subsequently added to the clause database of the DPLL solver. On the other hand the assignments returned from the DPLL solver guide the local search with a new start point. The synergies from both the local search and DPLL search were investigated. Experimental results demonstrated up to an order of magnitude performance improvement for hard satisfiable instances. Future research directions include WALKSAT guided preprocessing and alternative padding mechanisms. Due to the fact that HBISAT tries to accumulate a set of broken clauses in the clause database of the DPLL solver that might be unsatisfiable, further studies could be conducted on the potential to extract an UNSAT core using this hybrid approach.

References

[1] WALKSAT, http://www.cs.rochester.edu/u/kautz/walksat/.

[2] SAT Competition 2002, http://www.satlive.org/SATCompetition/.

[3] ZChaff, http://www.princeton.edu/~chaff/zchaff.html.

[4] Minisat, http://minisat.se/MiniSat.html.

[5] IBM Research, www.haifa.il.ibm.com/projects/verification/RB_Homepage/bmcbenchmarks.html.

[6] S. Barner, D. Geist, and A. Gringauze. Symbolic localization reduction with reconstruction layering and backtrackin. In *Conference on Computer-Aided Verification*, pages 65–77, 2002.

[7] Hachemi Bennaceur, Idir Gouachi, and Gerard Plateau. An incremental branch-and-bound method for the satisfiability problem. *INFORMS J. on Computing*, **10**(3):301–308, 1998.

[8] Stephen A. Cook. The complexity of theorem proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[9] Martin Davis, George Logemann, and Donald W. Loveland. Machine program for theorem proving. *Communications of the ACM*, **5**(7):394–397, 1962.

[10] Martin Davis and Hillary Putnam. Computing procedure for quantification theory. In *Journal of the ACM*, **7**, pages 201–215, 1960.

[11] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Proceedings of the 6th International Conferences on Theory and Applications of Satisfiability Testing*, pages 502–518, 2003.

[12] Hai Fang and Wheeler Ruml. Complete local search for propositional satisfiability. In *Proceedings of 19th National Conference on Artificial Intelligence*, pages 161–166, 2004.

[13] Brian Ferris and Jon Froehlich. Walksat as an informed heuristic to dpll in sat solving. http://www.cs.washington.edu/homes/bdferris/papers/WalkSAT-DPLL.pdf.

[14] Alex S. Fukunaga. Efficient implementations of sat local search. In *Posters of the 7th International Conferences on Theory and Applications of Satisfiability Testing*, 2004.

[15] Djamal Habet, Chu Min Li, Laure Devendeville, and Michel Vasquez. A hybrid approach for sat. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, pages 172–184, London, UK, 2002. Springer-Verlag.

[16] Panagiotis Manolios and Sudarshan K. Srinivasan. A parameterized benchmark suite of hard pipelined-machine-verification problems. In *Proceedings of 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 363–366, 2005.

[17] Bertrand Mazure, Lakhdar Sais, and Eric Gregoire. Boosting complete techniques thanks to local search methods. *Annals of Mathematics and Artificial Intelligence*, **22**(3-4):319–331, 1998.

[18] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535, 2001.

[19] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 337–343, Seattle, 1994.

[20] Bart Selman, Hector J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 440–446, 1992.

[21] Ofer Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. In *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 58–70, 2001.

[22] Miroslav Velev. http://www.ece.cmu.edu/~mvelev/sat_benchmarks.html.

[23] Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formulas. In *Presentations of 6th International Conferences on Theory and Applications of Satisfiability Testing*, 2003.