A New Hybrid Solution to Boost SAT Solver Performance *

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

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 to be passed to a DPLL SAT solver through an incremental interface. 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 can be obtained via the proposed hybrid solver.

1 Introduction

Boolean Satisfiability (SAT) has a wide-spread application domain, including computer aided design, artificial intelligence, planning, etc. Various Electronic Design Automation (EDA) applications 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 [2], many instances still remain extremely difficult.

A SAT problem takes as input a propositional formula that is often represented as the 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 a systematic search-tree based and the other is stochastic local-search based. The classical DPLL [3] algorithm and its derivatives like Chaff [11], Minisat [4] are all search-tree based. In a search-tree-based algorithm, a variable is selected and assigned at each step. This assignment will be applied onto the formula, through which new assignments of variables may be implied. If the current assignment to the variables causes a conflict, the solver will backtrack and make a different decision. In the process, conflict analysis can be performed to yield additional knowledge to aid future searches. These steps are repeated until either a solution is found or no solution is proved. On the other hand, local-search solvers such as GSAT [14] and WALKSAT [13] 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 evaluated to false). A number of steps of local greedy search are followed 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, the set of the unsatisfied clauses is hoped to become empty and thus a solution can be found. 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. To avoid being trapped indefinitely at local optimal points, 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 (note that Fang had published a complete local search algorithm [5]). Although local search algorithms can perform faster on a number of applications, it is hindered by its incompleteness.

Because both DPLL search and local search have their own strengths, there has been various attempts to combine them [8] [10]. In [10], a local search is used to help the DPLL-based 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 [6]. In [8], WALKSAT is used to exploit the variable equivalences and dependencies at certain nodes of the DPLL tree.

In this paper we propose a new framework to integrate local search and DPLL search. This integration does not change the decision order in the underlying DPLL SAT solver, and it offers completeness in the search. A local search is used to identify a subset of clauses to be passed to a DPLL SAT solver through an incremental solver interface [1]. 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 local optimal points. In order to take advantage from both the local and DPLL search strategies, several features are proposed to make the integration seamlessly. For satisfiable instances, up to an order of magnitude speedup can be obtained via the proposed

*supported in part by NSF Grants 0305881 and 0417340

978-3-9810801-2-4/DATE07 © 2007 EDAA

hybrid solver.

The remainder of the paper is organized as follows: The next section describes the preliminaries of DPLL and WALKSAT algorithm. The details behind our hybrid incremental SAT solver are presented at Section 3. Section 4 discusses the synergies between the DPLL and WALKSAT via some case studies. Section 5 presents the empirical evaluation of our proposed SAT solver. Our work is concluded in Section 6.

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, Logeman and Loveland, and it is the basis for most modern SAT solvers. The pseudo-code of DPLL is listed in Listing 1 [11].

Listing 1. DPLL Algorithm

```
def DPLL
begin
    while (true)
        if (!decide()) /*if no unassigned vars*/
            return SAT
        while (!bcp())
            if (!ResolveConflict())
                return UNSAT
end
def ResolveConflict()
begin
    d = most recent decision not tried bothways
    if (d == NULL) // no such d was found
        return false;
    flip the value of d;
    mark d as tried both ways;
    undo any invalidated implications;
    return true;
end
```

The decide() function in the DPLL algorithm incorporates the decision order heuristic. The most time-consuming part is the bcp() function, where the formula is evaluated under the current variable assignments to check if any other variable assignments may be implied, or that a conflict has been encountered. If the formula is satisfiable, an assignment, A, to the variables will be generated.

WALKSAT [13] was proposed in 1994 as an improvement of the previous GSAT [14]. First, several definitions will be explained before going deeper in the WALKSAT algorithm, shown in Listing 2. A clause is said to be *broken* if all the literals in this clause are evaluated to be 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 for flipping the given variable. When a broken clause becomes non-broken by flipping the value of a variable, usually it is referred to as this clause is *healed* by this 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 $(b+c)$ is left as *broken*. Now the broken-counts of variable $\{a, b, c\}$ are $\{1, 0, 1\}$.

Listing 2. WALKSAT Algorithm

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

In the WALKSAT algorithm, 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 the chosen to be flipped. This can be implemented in an efficient way like two-literal watching scheme in Chaff [7]. WALKSAT will try a preset number of steps (CUTOFF) before it claims that it has failed to find a solution.

3 Our Approach

In order to combine the powers of both local search and DPLL-based search, the previous approaches mainly tried to embed the local search result into a DPLL-based SAT 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 are 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 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 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 Listing 3. 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. If the local search cannot find a solution within a given 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. 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 $B_i \bigcup C_{i-1}$ is UNSAT in any iteration i, then the original formula is guaranteed to be UNSAT; this allows for early termination of the SAT search. 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. 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 duplication of broken clauses, in our implementation, each clause carries a flag to indicate whether it has been added or not. Another advantage of employing an incremental SAT solver is that conflict clauses obtained can be carried from one iteration to the next [15] [17].

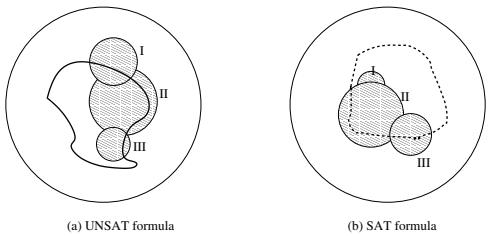
Listing 3. HBISAT Algorithm

```
def HBISAT_Solver(F/*input formula*/)
begin
    C=∅ /*Set of broken clauses*/
    A=∅ /*Assignment*/
    i=0 /*iteration counter*/
    CUTOFF=MAX_LOCAL_SEARCH_STEP
    while (true)
        if (A==∅)
            RandomInitalWalkSAT()
        else
            InitialWalkSAT(A)
        Status=WALKSAT_Solver(CUTOFF)
        if (Status==SAT)
             return SAT
        else if (Status==UNKNOWN)
            Result=CallDPLL(i)
            if (Result==SAT)
                A=GetAssignmentFromDPLL()
            else /*UNSAT or out of memory*/
                 return Result
    i++ /*count interation*/
end
def CallDPLL(i)
begin
    B=GetBrokenclause()
    /*guarantee one more clause will be added*/
    B = B ⋃ RandomPickOneNewClause()
    C = B ⋃ C
    AddClauseToDPLLSolver(C)
    Status=DPLL_Solver()
    return Status
end
```

Theorem 1. HBISAT is complete

Proof. In the HBISAT Algorithm shown in Listing 3, whenever B_i from iteration i is empty, then a satisfying solution has 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. With B_i added into the clause database of the DPLL solver, after a finite number of iterations the DPLL solver will eventually contain the entire original formula. Because the DPLL solver is complete, we can conclude that HBISAT is also complete. □

Figure 1 illustrates how HBISAT works on SAT formulas. Figure 1(a) shows the scenario that the formula is unsatisfiable and Figure 1(b) is for the scenario when the formula is satisfiable. In both parts (a) and (b) of the figure, the outer-most 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



(a) UNSAT formula (b) SAT formula

Figure 1. HBISAT on SAT & UNSAT formulas

three different broken clause sets generated from three different iterations.

If the original formula is unsatisfiable, there exists an unsatisfiable core (assuming there is only one UNSAT core) [18]. This core is represented by the region enclosed by the thicker line in part (a) of the figure. We can observe that each broken clause set covers portions of the unsatisfiable core. In each iteration of HBISAT, a portion of the unsatisfiable core will be identified. With the increasing number of iterations, the entire unsatisfiable core will be extracted from the formula which leads to an early termination.

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 the figure shows such a scenario where the region enclosed by the dashed line represents a hard-to-simultaneously-satisfy set. With the underlying assumption that WALKSAT can find the solution for those easy clauses outside the hard-to-simultaneously-satisfy region, the hard-to-simultaneously-satisfy sets will be identified by HBISAT gradually in a similar way as to 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 formula.

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 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 belongs to UC, i.e. $B \bigcap UC = \emptyset$. This means that WALKSAT must have obtained an assignment that satisfies all clauses outside of B, where B is the only broken clause set returned by WALKSAT. However, because $B \bigcap UC = \emptyset$, the clauses outside B contains the complete UC, 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 an incremental SAT solver, these hard spots will be solved incrementally and the conflict clauses learned through them have tremendous potential to help finally solving the formula.

HBISAT actually is a flexible SAT framework. Instead of the specific local search and DPLL solvers we used in this paper, other engines can easily be plugged in. The interaction between our local-search solver and DPLL-search solver is simply through a uniform calling interface, which is supported by most modern SAT solvers. The requirements for the DPLL based solver is that it should support a 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 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, it may find the unsatisfiability in the DPLL solver 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 clause padding procedure: 1) Based on the assumption that the flip frequency of a variable usually indicates its importance of solving the formula, the clauses which contains the most frequently flipped variable are padded into the clause database of the DPLL solver. 2) We put all the literals of broken clauses into a 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 to the broken clause set, with the hope that they will help to constrain the SAT solver.

4 Study of Synergies

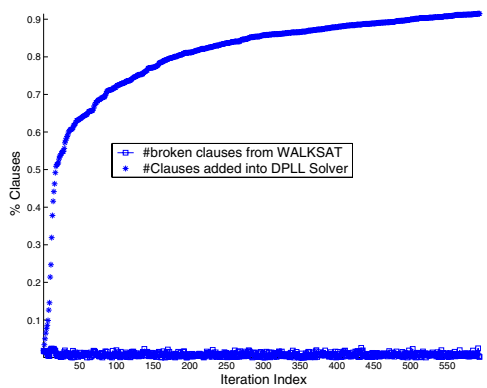


Figure 2. Early termination of an UNSAT instance

In this section, we will explore the interactions between WALKSAT and DPLL solver through some examples. For Figures 2, 3, and 4, the X-axis denotes the iteration index and the Y-axis denotes the percentage of clauses. In Figure 2, a early termination case is shown. 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 has been added, the DPLL solver can conclude that the instance is UNSAT. Although it does not necessarily mean the current clauses in the DPLL solver form a minimal UNSAT core, the early termination provides the potential to reduce the computational effort.

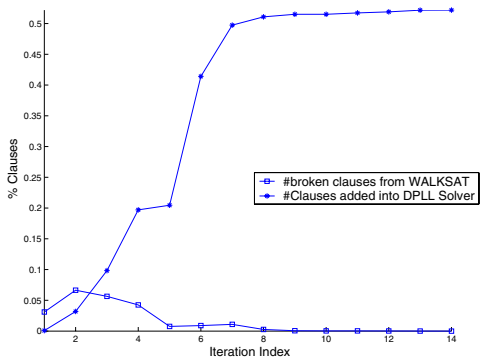


Figure 3. Solution found at an early stage by WALKSAT

Figure 3 presents an example that WALKSAT becomes more effective when guided by the solution returned from the DPLL solver. One can observe that the BRK curve generally drops with the increasing number of iterations. This is because the partial assignments supplied from the DPLL solver gives a better guide to the WALKSAT. Another interesting phenomenon is 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 the 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 the WALKSAT for this SAT instance. In contrast, the pure WALKSAT cannot obtain the solution for this instance after 200 iterations with the same CUTOFF value.

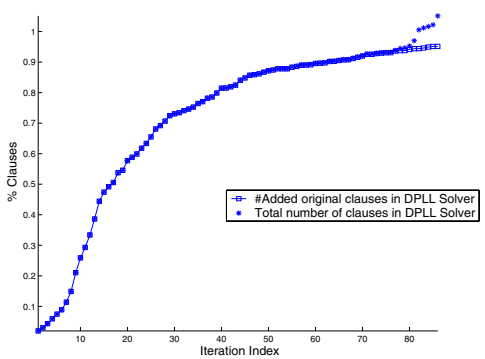


Figure 4. Example of gradually learning

If the early termination is not achievable and a solution cannot be found by WALKSAT, the entire instance will eventually be added into the DPLL solver. Figure 4 illustrates how

the conflict clauses help to finally solve the instance. 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 learning. After 90 iterations, more than 10% of the clauses in the DPLL solver are conflict clauses. At this point, when we add all the clauses from the original instance to the DPLL solver it takes 4.48 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.

5 Experimental Results

The proposed HBISAT was implemented in C++ under the Linux operating system, and experiments were conducted on a Pentium-4 3.2G PC with 1GB RAM. To demonstrate the portability of our algorithm, HBISAT was built on top of two popular SAT solvers, ZChaff version 2004.11.15 Simplified and Minisat 1.14. 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 [12]. 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 [9], 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 [16]. 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 was used in place of ZChaff or Minisat, as the framework for HBISAT requires only that the underlying DPLL-based SAT 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.

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. Finally, "OUT" indicates that the SAT solver aborts after a preset limit is met, which is 3000 seconds in our experiments.

It can be observed that among all the twenty-four Category-I instances, ZChaff aborted on four of them while HBIz could solve every instance within 2000 seconds. HBIz outperformed ZChaff in 9 of the 12 instances. For the other three 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 8 of 12 instances. For the UNSAT instance 07.k80, HBIm only took 29.39 second while Minisat's running time is 20 times that.

Next, experiment II was set up to evaluate the unsatisfiable instances, with the results reported in Table 2. The first four columns of Table 2 are similar as in Table 1. The run time of Minisat and HBIm were reported in the last two columns. The performance of HBISAT was comparable to the DPLL solver for most instances. ZChaff and HBIz aborted on exactly two instances and the average run times for the other instances were nearly equal. Meanwhile, Minisat and HBIm aborted on three instances. Generally HBIm gives better performance on harder instances thus leads to a total 500 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

Table 1. Category I Benchmarks

Instance	#V/#C	SAT?	ZChaff(s)	HBIz(s)
01.k40	29249/124460	SAT	158.893	**102.58**
01.k50	36339/154810	SAT	OUT	**556.91**
01.k60	43429/185160	SAT	OUT	**917.46**
04.k40	46225/198298	SAT	124.62	**92.54**
04.k50	58595/251378	SAT	OUT	**414.46**
04.k60	70965/304458	SAT	OUT	**1648**
06.k40	49126/213666	SAT	118.21	276.83
06.k50	61776/268886	SAT	OUT	**1731.71**
06.k60	74426/324106	SAT	OUT	**1989.62**
07.k40	13151/35904	UNSAT	5.89	20.59
07.k50	15221/40774	UNSAT	5.89	19.19
07.k60	17291/45644	UNSAT	69.83	**45.02**
Instance	#V/#C	SAT?	Minisat(s)	HBIm(s)
01.k70	50519/215510	SAT	59.05	**53.97**
01.k80	57609/245860	SAT	187.61	**95.03**
01.k90	64699/276210	SAT	331.447	**108.39**
04.k70	83335/357538	SAT	215.83	243.04
04.k80	95705/410618	SAT	420.2	**323.2**
04.k90	108075/463698	SAT	698.65	**367.93**
06.k70	87076/379326	SAT	38.55	109.09
06.k80	99726/434546	SAT	111.46	**110.98**
06.k90	112376/489766	SAT	161.99	**151.48**
07.k70	19361/50514	UNSAT	5.65	14.17
07.k80	21431/55384	UNSAT	773.6	**29.39**
07.k90	23501/60254	UNSAT	6.393	12.5

Table 2. Category II UNSAT Benchmarks

Instance	#V/#C	ZChaff	HBIz	Minisat	HBIm
c7b	26058/77128	238.32	**153.01**	57.14	60.48
c8n	53697/159595	487.57	622.67	142.768	163.6
c9b	36757/109045	608.63	**380.11**	251.69	**234.5**
f9n	185149/552412	1823.24	1925.78	OUT	OUT
g9n	54631/161950	361.08	**312.51**	61.49	**49.88**
g9b	59110/175387	195.27	381.55	42.24	52.53
g9idw	125885/371998	365.02	390.27	69.17	93.9
g9nidw	170918/506584	2705.64	**2216.42**	836.45	**357.16**
c10	17121/50803	10.05	13.65	13.22	25.72
c10b	43517/129265	617.53	**587.73**	541.66	**509.12**
c10bi	147116/437224	OUT	OUT	OUT	OUT
c10bid	291912/828039	OUT	OUT	OUT	OUT
Total		7155.08	**6609.42**	2015.828	**1546.89**

Table 3. Category III Benchmarks

Instances	#V/#C	SAT?	ZChaff(s)	HBIz(s)	Minisat(s)	HBIm(s)
fvp-sat3.0/pipe-64-4-bug01.cnf	35853/1021170	SAT	37.9	**8.27**	1.58	4.1
fvp-sat3.0/pipe-64-4-bug02.cnf	35853/1021171	SAT	OUT	**12.81**	OUT	OUT
fvp-sat3.0/pipe-64-4-bug03.cnf	35947/992674	SAT	4.33	14.27	0.93	3.88
fvp-sat3.0/pipe-64-4-bug04.cnf	35854/1012315	SAT	35.88	**11.16**	4.75	34.97
fvp-sat3.0/pipe-64-4-bug05.cnf	35853/1022271	SAT	823.53	**40.56**	OUT	OUT
fvp-sat3.0/pipe-64-4-bug06.cnf	35853/1022271	SAT	253.97	**6.67**	OUT	**9.21**
fvp-sat3.0/pipe-64-4-bug07.cnf	35853/1022271	SAT	916.74	**10.19**	OUT	**110.77**
fvp-sat3.0/pipe-64-4-bug08.cnf	35622/1003074	SAT	36.03	**6.44**	OUT	**6.35**
fvp-sat3.0/pipe-64-4-bug09.cnf	35726/1011764	SAT	0.4	5.34	164.34	**6.05**
fvp-sat3.0/pipe-64-4-bug10.cnf	35839/1012135	SAT	0.57	3.62	528	**4.02**
fvp-sat3.0/pipe-64-4-bug11.cnf	35853/1012271	SAT	1152	**127.37**	2.63	6.9
Total			3261.35	**246.7**	702.23	**186.25**
fvp-unsat2.0/5pipe.cnf	9471/195452	UNSAT	22.77	31.28	86.68	203.82
fvp-unsat2.0/6pipe.cnf	15800/394739	UNSAT	169.9	**126.97**	OUT	OUT
fvp-unsat2.0/6pipe-6-ooo.cnf	17064/545612	UNSAT	292.6	301.51	230.49	235.6
fvp-unsat2.0/7pipe.cnf	23910/751118	UNSAT	386.87	408.06	OUT	OUT

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 916.74 seconds, HBIm cost roughly 110 seconds while HBIz needed only 10.19 seconds. The power of combining local search and DPLL search is evident via both experiments I, II and III. In terms of UNSAT instances in 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.

6 Conclusions

In this paper, a new Hybrid SAT solver framework (HBISAT) has been presented that combines the power of local search and modern DPLL-based search with conflict-driven learning. In HBISAT, the local search instructs a DPLL SAT solver through an incremental solver interface. The synergies from both the local search and DPLL search are investigated. In effect, our guided local search identifies incremental sets of clauses that are hard, and these clauses are subsequently added to the clause database of the DPLL-based solver. Experimental results demonstrated that up to an order of magnitude performance improvement has been achieved for the hard satisfiable instances. Future research directions include WALKSAT guided preprocessing and alternative padding mechanisms.

References

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

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

[3] M. Davis, G. Logemann, and D. W. Loveland. Machine program for theorem proving. In *Communications of the ACM*, volume 5, pages 394–397, 1962.

[4] N. Eén and N. Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, 2003.

[5] R. W. Fang H. Complete local search for propositional satisfiability. In *Proc. of 19th National Conference on Artificial Intelligence*, pages 161–166, 2004.

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

[7] A. S. Fukunaga. Efficient implementations of sat local search. In *SAT*, 2004.

[8] D. Habet, C. M. Li, L. Devendeville, and M. Vasquez. A hybrid approach for sat. In *CP '02: Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, pages 172–184, London, UK, 2002. Springer-Verlag.

[9] P. Manolios and S. K. Srinivasan. A parameterized benchmark suite of hard pipelined-machine-verification problems. In *CHARME*, pages 363–366, 2005.

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

[11] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[12] I. Research. http://www.haifa.ibm.com/projects/ verification/rb_homepage/bmcbenchmarks.html.

[13] B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, pages 337–343, Seattle, 1994.

[14] B. Selman, H. J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In P. Rosenbloom and P. Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. AAAI Press.

[15] O. Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. *Lecture Notes in Computer Science*, 2144:58–70, 2001.

[16] M. Velev. http://www.ece.cmu.edu/∼mvelev/sat_benchmarks.html.

[17] J. Whittemore, J. Kim, and K. A. Sakallah. Satire: A new incremental satisfiability engine. In *DAC*, pages 542–545, 2001.

[18] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formulas. In *SAT*, May 2003.