

Exploring Constraint Satisfiability Techniques in Formal Verification

Lei Fang

Dissertation submitted to the Faculty of
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Electrical Engineering

Dr. Michael S. Hsiao, Chair

Dr. Sandeep K. Shukla

Dr. Chao Huang

Dr. Yilu Liu

Dr. Ezra A. Brown

May 06, 2008

Blacksburg, Virginia

Keywords: SAT, MIN-ONE SAT, MAX-SAT, Hybrid, Relaxation,
Double-Layer Conflict Driven Learning

Copyright © 2008, Lei Fang

Exploring Constraint Satisfiability Techniques in Formal Verification

Lei Fang

Abstract

Due to the widespread demands for efficient Propositional Satisfiability (SAT) solvers and its derivatives in Electronic Design Automation applications, methods to boost the performance of the SAT solver are highly desired. This dissertation aims to enhance the performance of SAT and related SAT solving problems. A hybrid solution to boost SAT solver performance is proposed as an initial attack in this dissertation, via an integration of local and DPLL-based search approaches.

Next, a different hybrid strategy is attempted that takes advantage of the conflicts in the SAT search, which plays a critical role in modern SAT solvers. Usually a learned conflict-induced clause is added back to the clause database. Although conflict-induced clauses help to block a portion of the search space, they can also become a burden due to the added cost in memory consumption and Boolean Constraint Propagation (BCP). We thus propose a novel double-layer conflict-driven learning to store only those “primary” conflict clauses back into the clause database while keeping the other clauses as pseudo Boolean constraints. With this approach our experiments demonstrate that the approach can improve both in performance and memory consumption. This work opens the door on how to assess the usefulness of conflict induced clauses.

Besides the aforementioned works about enhancing SAT solver performance and reducing memory cost, this dissertation also proposed a contributing work on the extended SAT problem solving. The current SAT solvers can provide an assignment for a satisfiable propositional formula. However, the capability for a SAT solver to return an “optimal” solution for a given objective function is severely lacking. MIN-ONE SAT is an optimization problem which requires the satisfying assignment with the minimal number of Ones, and it can be easily extended to minimize an arbitrary linear objective function. While some research has been conducted on MIN-ONE SAT, the

existing algorithms do not scale very well on large formulas. This dissertation presents a novel approximation algorithm (RelaxSAT) for MIN-ONE SAT. RelaxSAT generates a set of constraints from the objective function to guide the search. The constraints are gradually relaxed to eliminate the conflicts with the original Boolean SAT formula until a solution is found. The experiments demonstrate that RelaxSAT is able to handle very large instances which cannot be solved by existing MIN-ONE algorithms; furthermore, RelaxSAT is able to obtain a very tight bound on the solution with one to two orders of magnitude speedup.

Based on the proposed powerful MIN-ONE SAT algorithm, we built a MAX-SAT solver which achieved more than one order of magnitude speed up compared with the-state-of-art MAX-SAT solver. We also discuss a promising application of this MAX-SAT solver in formal verification.

Acknowledgements

First and foremost I thank my advisor Dr. Michael S. Hsiao for his guidance and support in my PhD life. I am so grateful to get opportunities to discuss my research works with him and be able to obtain very insightful suggestions on my research direction. During the four years I stayed in Virginia Tech I enjoyed working with him and kept learning from him. Without the help from him, I can't imagine of finishing my PhD.

I would like to thank Dr. Sandeep Shukla , Dr. Chao Huang, Dr. Yilu Liu and Dr. Ezra Brown to serve on my committee and spend precious time on the dissertation review and oral presentation. I would like also to thank all the Proactive members for their suggestions on my research and dissertation.

Last but not the least, I would like to express my deep gratitude to my family members for their unconditional support and love. This dissertation is to all the people who made this work possible. I sincerely appreciate all the help.

Lei Fang

May, 2008

To my parents, Ying and my sons: Pingping and Fanfan

Contents

Table of Contents	vi
List of Figures	ix
List of Tables	x
1 Introduction	1
1.1 Contributions of This Dissertation	4
1.2 Organization of The Proposal	5
2 Background	7
2.1 Satisfiability Problem and Its Solving	7
2.2 Potential to Improve Conflict Driven Learning	13
2.3 Overview of MIN-ONE SAT: A SAT Optimization Problem	15
2.4 Overview of MAX-SAT: Another SAT Optimization Problem	17
2.5 Preliminaries on the Basic SAT Algorithms	20
2.5.1 DPLL and WALKSAT Preliminaries	20
2.5.2 MIN-ONE SAT Preliminaries	23
2.5.3 MAX-SAT Preliminaries	25
3 Hybrid SAT Solution	29
3.1 Our Approach	30
3.1.1 Clause padding	35
3.1.2 Enhance WALKSAT with conflict clauses from DPLL	35
3.2 Study of Synergies	36

3.3	Experimental Results	39
3.4	Summary	42
4	Double-Layer Conflict Driven Learning with Pseudo Boolean Constraints	46
4.1	Two-layer Conflict Driven Learning	47
4.1.1	Pseudo Boolean Constraint Set	47
4.1.2	Basic Algorithm	48
4.1.3	Recovery Hash Table	50
4.2	Evaluation of Conflict Induced Clause Usefulness Estimation	51
4.2.1	Evaluation Metrics	51
4.2.2	Evaluation Function	53
4.3	Experimental Results	55
4.4	Summary	58
5	A Fast Approximation Algorithm for MIN-ONE SAT	63
5.1	Our Approach	64
5.1.1	RelaxSAT	64
5.1.2	Relaxation Heuristic	66
5.1.3	Discussion on Complexity of RelaxSAT	69
5.2	Experimental Results	69
5.3	Summary	73
6	A RelaxSAT based MAX-SAT Solver	77
6.1	The New MAX-SAT Solver	78
6.2	Evaluation of MAX-SAT Solver	79
6.3	Applications of RMAXSAT in Bounded Model Checking	82
6.3.1	Discussions on RMAXSAT_ALL	87
6.4	Summary	88
7	Summary of This Dissertation	92

List of Figures

2.1	An example of implication graph	10
2.2	Conflict-driven learning	11
2.3	MIN-ONE SAT based MAX-SAT Solving	26
3.1	HBISAT on SAT & UNSAT formulas	32
3.2	Abstraction and refinement in HBISAT	34
3.3	Early termination of an UNSAT instance	37
3.4	Solution found at an early stage by WALKSAT	38
3.5	Example of gradually learning	39
3.6	Broken clauses distribution returned by WALKSAT	40
4.1	Double-layer conflict driven learning algorithm	48
4.2	Conflicts checking traditional vs. with PBCS	49
4.3	Recovery hash table working flow	51
5.1	Exploring space enlargement	67
5.2	Conflict guided relaxation heuristic	68
5.3	Memory consumption: OPTSAT vs. RelaxSAT	73
5.4	Memory consumption: MiniSAT+ vs. RelaxSAT	74
6.1	Basic Bounded Model Checking Setup	82
6.2	Enhanced Bounded Model Checking Setup	83

List of Tables

3.1	Category I Benchmarks	43
3.2	Category II UNSAT Benchmarks	44
3.3	Category III Benchmarks	45
4.1	Rules of Usefulness Metrics	54
4.2	Importance of Individual Evaluation Metrics	59
4.3	General SAT Instances Results	60
4.4	Experiments on Group of Instances That Shares Similarities	61
4.5	Instance List	62
5.1	Experiment I: OPTSAT Benchmarks	75
5.2	Experiment II: Power-Aware Test Generation Benchmarks	76
5.3	Experiment III: Objective Functions with Integer Coefficients	76
6.1	Existing MAX-SAT Comparison	89
6.2	RMAXSAT vs. OPTSAT	90
6.3	RMAXSAT vs. MSAT+	91

Chapter 1

Introduction

Since its birth in the 1950s, the semiconductor industry has changed the way people work, communicate and live. From consumer electronics to high performance supercomputers, a vast majority of the devices rely on some form of digital circuits. Although the current Very Large Scale Integration (VLSI) chip design and manufacturing industry is already a multi-billion dollar industry, it still holds an immeasurable potential to grow. With today's advanced technology, engineers are able to design chips with several hundreds of millions of transistors. For such huge designs, assuring their correctness is even harder than their actual design. Consequently, design verification plays a central role in the design flow due to the fast development of VLSI technology. It has been reported that over 70% of the total design effort needs to be allocated allocated to verification [25][64]. This problem is especially severe in the deep sub-micron era due to rising design complexity. The core value of design verification is to help the designers to ensure the correctness of the design. Furthermore, verification can enhance the robustness of the design because it helps check the corner cases which may not be carefully considered in the early design stages. Such confidence in the correctness of the design is vital in many critical applications. For instance, when designing a traffic light controller, the design must guarantee that two perpendicular green lights cannot be simultaneously on. Verification tools must be able to prove that such scenarios are *impossible* to happen in the design. Typically it is less costly to fix bugs in the early stages

and hence deploying verification tools also helps to reduce the development cost. The criticality of verification necessitates the development of more efficient verification techniques.

In order to ensure that verification goals are met, various techniques have been developed over the years. Currently the state-of-the-art verification tools can handle large designs with millions of gates with reasonable computation and resource cost. These verification approaches can be roughly divided into two categories. The first is simulation-based and the second is commonly referred to as formal verification. Simulation-based techniques try to simulate a large portion of the entire input space that a design may encounter to see whether it can function correctly [60][77]. Simulation-based verification has been widely adopted in industry over the decades due its low overhead and easy implementation. When the size of the design increases, it becomes unrealistic to enumerate all the possible scenarios. But the basic rule still holds: the more scenarios you simulate, the higher confidence you have in the correctness of the design. In order to increase the confidence of the verification results, billions of scenarios have to be tested for today's real designs which certainly becomes a unbearable burden in terms of budget and time-to-market. Given this fact, even though the current simulation-based verification techniques have been developed and pruned to be very intelligent, they still lag behind the demands of real designs. The biggest challenge of simulation-based verification is that it cannot guarantee the correctness unless all the scenarios are exhaustively simulated, which is frequently referred to as the "coverage problem". The coverage here represents how much of the total scenarios have been covered by the simulation. Obviously in simulation-based verification higher coverage is always desired, but how to reach a desirable coverage rapidly remains a challenge.

To overcome these limitations, researchers and engineers have turned to formal verification to seek alternative verification solutions. The term "Formal" in formal verification means that these techniques can guarantee the assertions made and no corner cases will be missed. Usually formal verification methods are complete because they perform an implicit and complete search on all the possible scenarios. Formal verification techniques significantly boost designers' capability to find hard-to-find bugs and totally eliminate the coverage problem. Since in formal verification we have to implicitly search all the scenarios it is not surprising that the backbone

solvers have to be extremely intelligent and powerful. Binary Decision Diagrams (BDD) [20] have been a widely used structure in formal verification techniques. They are included in tools such as [44][22] [10][21]. BDDs provide a way to store Boolean functions compactly and manipulate them efficiently. Using BDDs, it is theoretically possible for us to handle complex designs directly and conduct the verification steps in a complete manner. However, since BDD-based techniques store the complete representation of the design in a canonical fashion, they face the memory blowup problem. Although this problem can be eased by BDD optimization it cannot be avoided in some cases. So, recently, researchers have started work on those formal verification techniques that are not memory-hungry.

In the last decade, another formal verification technique has emerged and has had a huge impact on the development of formal verification tools due to the huge strides made in propositional satisfiability (SAT). By modeling the verification problem through this well-known SAT formulation, we are able to take advantage of exceptionally potent SAT solvers to solve the verification problem. Both the SAT solver itself and its relevant formal verification problems have since become active research areas. In fact, SAT solvers are among the most vibrant research topics in formal verification today. Compared to BDDs, SAT is not as memory intensive but is still highly computationally intensive. By balancing memory size and CPU power, a good verification system may be a blend of both BDD and SAT techniques.

We will briefly show how the SAT based formal verification works by an example of equivalence checking [72][39]. In formal verification, equivalence checking is used to check whether two designs are functionally equivalent. Note that a functionally equivalent design is not necessarily structurally equivalent. One may see that equivalence checking is an ideal application of formal verification where 100% confidence is mandatory. A classic SAT-based equivalence checking can be done in the following way: Given two designs to be checked, their inputs are tied together and their outputs are monitored to see whether there exists an input pattern that can generate a distinguished output. This setup can be transformed into a satisfiability instance in linear time. After we obtain this SAT instance, a SAT solver is invoked to solve it. The SAT solver should be able to declare this instance as satisfiable or unsatisfiable, which determines if the two candidate de-

signs are equivalent or not. SAT-based formal verification is also used in other formal verification, such as Bounded Model Checking (BMC) [28][58], Unbounded Model Checking (UMC) [57][46], Reachability Analysis [4][41], etc. Usually the conversion of a problem to a SAT instance is relative easy, the difficult part is the solving of the instance. As the SAT solver now is the backbone of many formal verification techniques, a high-performance SAT solver is highly craved.

As mentioned before, significant research has already been conducted on SAT solvers, from which many efficient and scalable techniques and heuristics have been discovered. When aiming to build a better SAT solver, it is wise to find a solution that can be benefited from all prior discoveries. After carefully analyzing the existing SAT solver algorithms, we propose a novel hybrid framework to boost the SAT solver performance, especially for formal verification problems. This framework enhances the SAT solver performance to a higher level by taking advantage of existing SAT algorithms. We also plan to extend the hybrid approaches to some SAT related optimization problems and other formal verification problems.

In this dissertation we not only proposed a flexible framework to leverage the SAT solver performance by integrating the existing SAT solver algorithm, but also pursued to improve the fundamental SAT solver techniques. Furthermore, we conducted extensive research on SAT optimization problems, such as MIN-ONE SAT[38]and MAX-SAT[36].

1.1 Contributions of This Dissertation

Although SAT solvers have been intensively studied and widely used in both industry and academia, the demand for more powerful SAT solvers has never been greater. In this dissertation, firstly a hybrid framework that integrates two orthogonal SAT algorithms is presented. This hybrid solution takes strength from both sides while alleviating their weakness by interacting with each other. This combination is not-trivial and is entirely different from all the existing hybrid attempts. This hybrid framework uses a abstraction-refinement flow and can be flexibly deployed on top of current SAT solvers. The experimental results demonstrate its effectiveness and flexibility.

To further leverage the performance of SAT solver and even reduce the memory cost simultaneously, we investigated one of the most important fundamental SAT solving techniques-conflict driven learning, which leads to our second major contribution of this dissertation: double-layer conflict driven learning. Our proposed double-layer architecture allows extra flexibility to store the conflict-induced clause. Meanwhile it can bring significant performance boost due to a better decision order and reduced Boolean constraint propagation cost by suppressing the supplementary conflict-induced clauses.

Although the SAT solver is able to handle many applications from broad application domains, generally it lacks the capability to provide a optimized solution. Currently more applications are emerging to demand an accurate yet fast algorithm on this kind of SAT optimization problems: satisfiability combined with additional objective functions. But in reality all the existing algorithms fall short due to the unbearable computational complexities. Our third major contribution is the first approximation algorithm even known in this area, which reduces the computational complexity radically and still provides a tough bound. We believe with the help of our algorithm now it becomes feasible to solve many problems that is hardly possible in the past.

Finally, to further demonstrate the effectiveness of our MIN-ONE SAT algorithm, we built a MAX-SAT solver on top of this algorithm. This MAX-SAT solver exhibits more than one order of magnitude performance improvement when compared with the other best-in-clause MAX-SAT solvers.

1.2 Organization of The Proposal

The remainder of the dissertation is organized as follows: The next chapter describes the preliminaries of various SAT and MIN-ONE SAT algorithms. The details behind our hybrid incremental SAT solver are presented in Chapter 3. The double-layer conflict driven learning will be discussed thoroughly in Chapter 4. Our novel MIN-ONE SAT approximation algorithm is presented in Chapter 5. A MAX-SAT Solver and its application in formal verification is discussed in Chapter 6. All

the works are summarized in Chapter 7.

Chapter 2

Background

2.1 Satisfiability Problem and Its Solving

SAT is one of the most intensively studied problems due to its broad use in Artificial Intelligence (AI), planning, circuit testing and circuit verification, and other electronic design automation problems. Since the 1960s, various SAT techniques have been proposed.

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 solution to this formula is defined as a set of variable assignments that makes this formula *true*. In other words, all the clauses of this formula should be evaluated as true under such an assignment. If such a solution exists for a given formula, the corresponding SAT problem is said to be satisfiable. Otherwise it is unsatisfiable. Here is an example of a formula in CNF: $(\bar{a} + b)(\bar{b})$. This CNF formula includes two clauses, where the first clause contains two literals and the second clause has only one literal. There are two variables in this CNF, and a literal is a variable in either positive or negative polarity. Obviously this formula can be satisfied with the assignment $\{a = 1, b = 0\}$.

The first systematic approach to tackle the SAT problem is the DP algorithm [55]. In the DP

algorithm, a procedure called *binary resolution* is invoked repeatedly on the clauses of the formula until no more clauses can be resolved or an empty clause is discovered. In the first case, it means this formula is satisfiable while in the second case, the derivation of an empty clause indicates that the formula is unsatisfiable. The resolution procedure is actually an operation to generate a new clause from two clauses if these two clauses share at least one common variable with opposite polarities. The new clause generated by resolution is an implication of the original two clauses, also called a resolvent of the two clauses.

For example, consider the following CNF formula $F: (\bar{a} + b)(\bar{a} + c)(\bar{b} + \bar{c})$. For simplicity these three clauses are noted as C_1, C_2 and C_3 here. By applying resolution between C_2, C_3 and C_1, C_3 , we can obtain two new clauses $(\bar{a} + \bar{b})(\bar{a} + \bar{c})$, named C_4 and C_5 . Now for these five clauses, C_1, C_4 and C_2, C_5 can be resolved, which lead to clause $C_7, (\bar{a})$. By checking all the possible clause pairs in the clause set $\{C_1, C_2, C_3, C_4, C_5, C_6, C_7\}$, we can find that no resolution can be further applied. Based on the conclusion we mentioned before, this CNF is satisfiable since no empty clause can be derived. In this example, assignment $\{a = 0, b = 0, c = 1\}$ satisfies the formula. Let us look at another example, $(a + \bar{b})(\bar{a})(\bar{b})$. Similarly we name the three clauses as C_1, C_2, C_3 . If C_1, C_2 are resolved, a intermediate clause (b) will be obtained. Now we have two clauses (b) and (\bar{b}) . It is easy to see that the resolution between these two clauses will result in an empty clause which means the original formula is unsatisfiable. From these two examples we can see that a large number of clauses can be generated by the resolution process even for small-sized formulas. Hence, the memory cost can easily be a burden for the DP algorithm.

Since resolution only introduces implied clauses from the original formula, it will not change the satisfiability of the original formula. It has been proved that a formula is guaranteed unsatisfiable if an empty clause is generated by resolution [55], because an empty clause indicates that an internal conflict relies in the formula. When any pair of clauses in the formula does not share any common literal with opposite polarities, the resolution process will stop and this formula can be concluded as satisfiable. This is due to fact that if no common variable with opposite polarities is shared between clauses, then all the clauses can be assigned to be true independently. Although the DP algorithm is complete, it is not practical because the number of new clauses generated by

resolution can increase exponentially with respect to the number of variables, causing a memory blowup.

Next, the DPLL algorithm was introduced by Martin Davis, George Logemann and Donald W. Loveland in 1962 [29]. In the DPLL algorithm, instead of performing explicit resolution on the clauses, each time one variable is selected and assigned a value and the Boolean Constraint Propagation (BCP) procedure is invoked to identify the assignments implied by this assignment. If the current assignment to the variables causes a conflict, i.e., one or more clauses evaluate to false, the solver will *backtrack* and make a different decision. Since the DPLL algorithm implicitly traverses the entire solution space, this algorithm is complete. Although the DPLL algorithm has the advantage that the memory cost is only $O(n)$, the computation time can be exponential in the number of variables. We will show how the DPLL algorithm works on the preceding formula F as well. Given formula F , suppose we first pick variable a and assign it value ONE. Now, in order to satisfy C_1 variable b has to be assigned ONE. Usually this is called “ b is implied as ONE by the assignment $a = 1$ ”. Similarly variable c is also implied as ONE. When $\{a = 1, b = 1, c = 1\}$ is assigned to the formula F , we can observe that C_3 is unsatisfied. A conflict now occurs which indicates our current assignment is wrong. Looking back, clearly only one decision has been made so far, which is $a = 1$ (b and c are implied by $a = 1$, they are not decisions). Based on the algorithm the most recent decision will be flipped, $a = 0$. With this new assignment $a = 0$, clause C_1 and C_2 are already satisfied. In clause C_3 , when variable b is picked and assigned a ZERO, F can be determined satisfiable. From this simple example, one may notice that the efficiency of the DPLL algorithm highly depends on the variable order from which the decisions are made. It has been shown that finding the optimal decision order is NP-complete, some heuristics have been demonstrated to be efficient and scalable. Besides the decision order, the implication operation, normally called Boolean Constraint Propagation (BCP), also has a significant impact on the performance of the DPLL algorithm.

The concept of conflict-driven learning is a ground breaking technique in modern SAT solver development [54]. We now provide a brief overview on how conflict-driven learning works and why it is useful. During the search of the DPLL algorithm, one may find that we may be doing

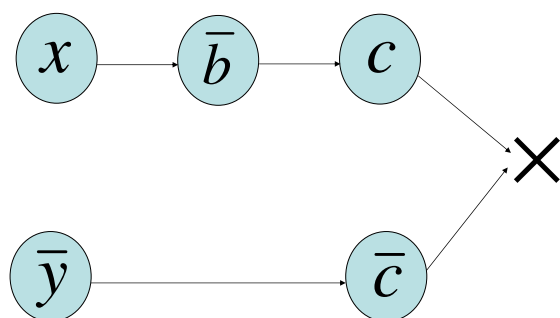


Figure 2.1: An example of implication graph

some unnecessary search. A simple example is shown in Fig 2.1, where a partial implication graph is shown. An implication graph illustrates the implication relationships between variables. In the implication graph, the vertices are the variables with their values, and the directed edges represent the implication relationships between the variables. In Fig 2.1, variable x with value ONE implies b equals ZERO which leads to c equals ONE. Meanwhile variable y implies c equals ZERO with assignment ZERO. Now we find that after assigning $x = 1$ and $y = 0$, a conflict arises because c is implied to two opposite values. Assume that this partial implication graph resides in a subtree T_A of the whole decision tree, as is shown in Fig 2.2. Due to conflict on variable c the solver will backtrack in this subtree. Suppose at a later time, the solver enters the search area of subtree T_B , shown in Fig 2.2. At that time, if variable x and y are still free variables (i.e., they have not been assigned a value or implied by previous assignments on other variables), it is highly possible that the solver tries to assign $x = 1$ and $y = 0$ which will definitely cause it to backtrack. One can see that this attempt of assigning $x = 1$ and $y = 0$ is unnecessary and should be avoided. Generalizing the preceding discussion, a technique can be developed to avoid unnecessary searches in the future by allowing the SAT solver to remember the history. This is the basic concept behind conflict-driven learning. The conflict-driven learning records the set of variable value combinations that will falsify the formula. The learned information is stored as

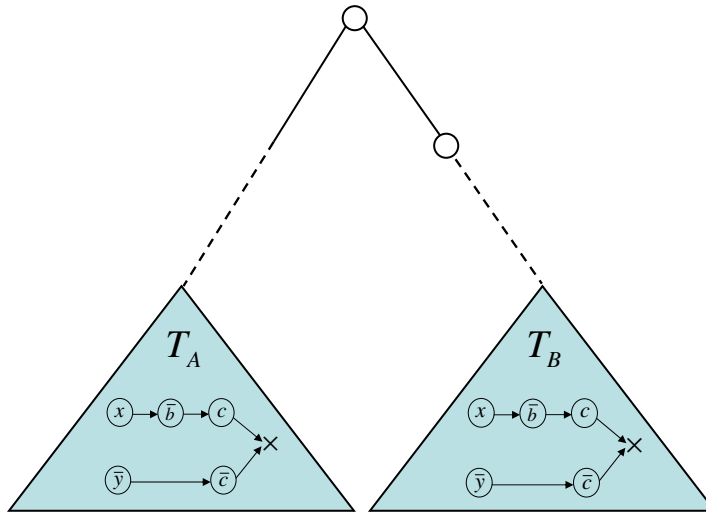


Figure 2.2: Conflict-driven learning

conflict clauses. These conflict clauses are added to the formula, gradually increasing its size with each added conflict clause. However, since the conflict clauses block the known futile assignments in advance, the computation resources are saved. In our example, the conflict-driven learning will add a newly learned clause $(\bar{x} + y)$ to the formula to prevent this futile assignment in the future. It should be noted that not only the assignments can derive the conflict-induced clauses, any cut in the implication graph can constitute the conflict learned clauses. These internal cuts are actually more powerful due to the fact that various assignments can be mapped to a same cut in the implication graph. Different learning strategies were discussed in detail in [79].

Conflict-driven learning helps to avoid unnecessary search, and in the mean time, it also introduces an overhead to the size of the formula. But the overall effect of conflict-driven learning is generally positive, as it can dramatically improve the performance and capability of modern SAT solvers. The introduction of conflict-driven learning brings the SAT application to a whole new level.

Both DP and DPLL algorithms are complete and deterministic, which means that given enough time and memory a formula can be proved unsatisfiable or a solution can be found. Despite their advantages, the DP and DPLL algorithm can suffer from either memory or temporal explosion. There exists another category of algorithms, which is light-weight and local search based. Usually people label these algorithm as stochastic search-based algorithms. GSAT [66] and WALKSAT [67] are two well-known algorithms in this category. In stochastic search algorithms, a complete assignment is generated 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 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.

Compared with stochastic local search, DPLL algorithm implicitly searches the entire search space. Thus it is referred to as the systematic search-tree based algorithm. Generally, search-tree based algorithms are complete while local search algorithms are incomplete (note that Fang had published a complete local search algorithm [34]).

It has been demonstrated that both DPLL search and local search have their own strengths and favored applications. There have been various attempts to combine them [43] [56]. In [56], 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 engine, the underlying decision order heuristics may potentially be degraded. This approach may not gain much performance as shown in [35]. In [43], WALKSAT is used to exploit the variable equivalences and dependencies at certain nodes of the DPLL tree.

2.2 Potential to Improve Conflict Driven Learning

The major breakthroughs in the SAT solver in the past few decades can be divided into (1) efficient Boolean Constraint Propagation (BCP), (2) well-tuned decision heuristic and (3) conflict-driven learning (CDL) [54][62]. In the SAT solving progress, CDL records the reasons of each conflict in an intelligent manner such that much of search space can be blocked by these added clauses in the subsequent searches. The usual implementation of CDL is the following: whenever a conflict arises, the current implication graph will be analyzed to produce a reason behind the conflict, yielding a conflict-induced clause (CIC). Since a CIC is constructed by the negation of those literals from a cutset of the implication graph, it is guaranteed that whenever this CIC is violated the solver can immediately backtrack. This is because all decisions after this point will only lead to conflict spaces. We can see that a CIC helps to block a portion of the search space in the search. The most popular heuristic to generate CIC from conflict is UIP (Unique Implication Point), which is described in detail in [81]. There have been other heuristics to analyze CIC [14][54].

Although CDL is a powerful technique to boost the SAT solver performance, it also has its limitations. One limitation is that the excessive number of conflict-induced clauses can significantly slow down BCP and consume extra memory. One may argue that each added conflict-induced clause is able to block some portion of the search space. While this argument is true, in reality not all conflict clauses may be useful. As a result, by adding every conflict-induced clause without checking if it is useful or not, it has been observed that many of the added clauses can only produce a very limited number of conflicts in its lifetime. Some conflict clauses also provide no additional pruning in the remainder of the search. We call these less useful conflict-induced clauses as “supplementary” clauses, in contrast to the useful ones as “primary” clauses. These supplementary clauses consume unnecessary computational resources. Worse, they bring noise to the decision ordering heuristic and reduce its effectiveness. Our motivation stems from this observation and we aim to find a solution that answers the following two questions: I) Can we improve the SAT solver by dealing with these supplementary clauses more efficiently? II) If so, how to efficiently identify these supplementary clauses?

One simple answer to question I is that these supplementary clauses should be dropped, thereby avoiding both the memory and BCP burden. While this strategy is quite simple, it may be overly optimistic and may mis-predict and discards some of those primary clauses that should have never been dropped. In regard to question II, mechanisms to overcome mis-predictions are needed. In this paper we propose a novel double-layer architecture to store the conflict-induced clauses, where the extra computational cost for supplementary clauses is minimal yet keeping a portion of their search space pruning capability. The first layer is the same as the conventional SAT solver, except that only those primary clauses are allowed to be added back to the clause database. In the second layer, the supplementary clauses are *compacted* and stored in a Pseudo Boolean Constraint Set (PBCS). PBCS is also checked regularly during the BCP and any conflict in PBCS will likewise invoke a backtrack. The advantage of introducing PBCS is that Pseudo Boolean Constraint (PBC) is more compact when representing the blocking information thus to significantly reduce the cost to represent those CICs[19][24]. We name our proposed technique as double-layer conflict driven learning.

It should be noted that many of current SAT solvers deploy techniques like database compaction [62][30] to periodically remove unused clauses. This is different from our approach since existing methods are passive while ours is not. Database compaction cannot be operated too frequently or it becomes ineffective. More importantly, the diluting effect of decision ordering caused by those supplementary clauses cannot be compensated by passive methods like database compaction. Furthermore, we do not remove the clause entirely, but these clauses are added to a pseudo Boolean constraint. In other words, the value of the less useful conflict clauses is not completely lost.

In our proposed double-layer architecture each pseudo Boolean constraint in the PBCS is computed as the summation of the assigned supplementary clauses. It should be noted that the summation operation often diminishes the blocking power of CICs, compared with the scenario where they were stored as distinct clauses in the first layer. This will be discussed in detail in Section XX. One can see that whether the double-layer technique can be successful or not decidedly depends on how accurate we can identify supplementary clauses. We will study the potential metrics to evaluate the usefulness of CIC in Section 4.2.

Chapter 4 describes the framework of double-layer conflict driven learning. Different approaches to evaluate conflict-induced clause are studied in Section 4.2. Section 4.3 presents the experimental results on our proposed techniques. Our works are concluded in the Section 4.4 with future work perspective.

2.3 Overview of MIN-ONE SAT: A SAT Optimization Problem

Although SAT is already widely used to target problems in various domains, more applications can be benefited by studying the derivations from basic SAT problem. Usually these problems can be modeled as a traditional SAT problem with additional objectives, such as finding a solution and achieving maximal or minimal values on the objective function simultaneously. Even though the recent successes in SAT have offered much promise, when facing the problems that requires an objective optimization, the current SAT solvers may not be applicable.

The heuristics embedded in the current SAT solvers generally steer the search to find the first solution as quickly as possible. Thus, for a satisfiable instance, one may argue that the solution returned by a SAT solver is an arbitrary one among the potentially huge number of satisfiable solutions. Furthermore, the "end-users" lack the ability to control the SAT solver to return a solution that they may favor. As a result, some applications are beyond the scope of the basic conventional SAT solver. For instance, consider the problem of generating an input sequence that can take a sequential circuit to satisfy some target such that the final state has the least Hamming distance from the initial state. When this sequence generation is modeled as a SAT problem on an unrolled instance of the circuit, the existing SAT solvers may be able to find a sequence that can satisfy the target objective, but they may have tremendous difficulty in obtaining the sequence that produces the final state with the least Hamming distance. Another related problem that requires an optimal solution is the problem of power-aware test generation, where the patterns generated should be generated with the minimum (or maximum) switching activities across two clock cycles [47][50]. Besides these examples, other applications are emerging to demand a more intelligent SAT solver which can return an optimal (or close to optimal) solution among its solution set. **Objective func-**

tions are commonly used to represent the optimality target. With appropriate simplification and normalization of the objective function, the objective function can often be represented as a linear function of the variable assignments. This puts the MIN-ONE SAT problem as an ideal candidate to address this class of problems.

The MIN-ONE SAT problem is defined as follows: Given a SAT formula, if it is satisfiable, find the variable assignment that contains the minimal number of ONES. MIN-ONE SAT can be viewed as a combination of the conventional SAT formulation and an optimization problem. It tries to find the best solution (minimal number of ONES in the assignment) among all the possible solutions for a satisfiable Boolean formula. Note that the MIN-ONE SAT is not meaningful for unsatisfiable SAT formulas, since an unsatisfiable formula will also be MIN-ONE unsatisfiable. Compared with the conventional SAT problem, MIN-ONE SAT demands more computation power because all the potential solutions need to be searched to identify the optimal one.

A naive approach to the MIN-ONE SAT problem is to compute all the possible solutions first, then identify the optimal one from among all the solutions. This approach requires an all-solution SAT solver, so the overhead may render it to be infeasible for some instances. Another way is to treat the MIN-ONE SAT problem as a special case of Pseudo Boolean SAT (PBSAT) problem [5][69]. The basic idea and procedure behind the PBSAT approach will be explained here briefly: Using the PBSAT concept, the minimal-ONES target can be viewed as searching the minimal sum of the assigned variable values. Suppose V_i is the value in the assignment of variable i , and suppose the total number of variables in the formula is N , then the to-be-minimized objective function for PBSAT is simply *minimize* $\sum_{i=1}^N V_i$. To find the minimal value, the PBSAT solver gradually tightens the minimal bound estimate until an optimum point is reached. In this approach it is easy to see that $\sum_{i=1}^N V_i$ can be replaced with any linear objective functions. More details about this technique will be covered in Section 2.5. In [38], a method called OPTSAT is proposed to handle MIN-ONE SAT via a modified DPLL [29] search algorithm. Although the incorporation between DPLL algorithm and MIN-ONE SAT optimization performs very well on some benchmarks, it lacks the flexibility to process arbitrary linear objective functions. Meanwhile the performance of the DPLL algorithm may be degraded because of a forced decision order. The current MIN-ONE SAT algorithms are

generally more suitable for small and medium sized formulas. For large MIN-ONE SAT formulas, they often abort, unfortunately.

Due to the added complexity of the MIN-ONE SAT problem, performing a complete search to find the optimal solution is extremely hard. This observation motivated us to propose an efficient approximation algorithm for MIN-ONE SAT, which should not only be fast, but have low computational overhead and high-quality approximation of the solution. In our approach, a set of constraints from the objective function is automatically generated to guide the search. This set of constraints is gradually relaxed to eliminate any conflict(s) with the original Boolean SAT formula until a solution is found. To the best of our knowledge, this is the first approximation algorithm targeting specifically on MIN-ONE SAT to achieve a tight bound on the solution. The experimental results show that our approach can obtain a tight bound when compared with existing complete-search based methods, with one to two orders of magnitude speedup.

The previous algorithms of MIN-ONE SAT will be discussed in Section 2.5.2. In Chapter 5, a novel approximation algorithm is proposed in Section 5.1. The experimental results are presented and explained in Section 5.2. The conclusions and future works will be laid out in the last section.

2.4 Overview of MAX-SAT: Another SAT Optimization Problem

Similar to MIN-ONE SAT, MAX-SAT is another important SAT optimization problem, both in theory and practice. Many difficult problems can be solved by MAX-SAT, such as Max-Cut, Max-Clique and Binary Covering Problem [9][36][15][59]. MAX-SAT is defined as follows: given a SAT instance, which is known to be unsatisfiable, find a variable assignment that satisfies the maximal number of clauses. Note that a clause is satisfied when at least one of its literals is evaluated to true. In other words, MAX-SAT optimizes the variable assignments to satisfy as many clauses as possible. One can see that the classic SAT problem can be viewed as a special case of MAX-SAT where all the clauses are satisfied.

First, let us take look at an example of the MAX-SAT. Suppose given a CNF formula $(a +$

$b)(a + \bar{b})(\bar{a} + c)(\bar{a} + \bar{c})$. This formula is unsatisfiable and have a MAX-SAT solution $a = 0, b = 1, c = 1$. Under this assignment, three clauses $(a + b), (\bar{a} + c), (\bar{a} + \bar{c})$ are satisfied with only clause $(a + \bar{b})$ evaluated to false. One may observe that assignment $\{a = 0, b = 1, c = 0\}$ can also satisfy three clauses to achieve the same bound as the previous assignment. Thus, for a MAX-SAT problem, the solution may not be unique.

Although MAX-SAT is also a SAT optimization problem, it is quite different from MIN-ONE SAT. In MIN-ONE SAT, all the constraints (clauses) have to be satisfied and a objective function optimized simultaneously. On the other hand, the MAX-SAT solver attempts to maximize the satisfiability of the formula. For satisfiable instances, the MAX-SAT solver should be able to return a SAT solution. On those unsatisfiable instances, the MAX-SAT solver should be able to tell the maximal number of clauses that can be satisfied and the corresponding variable assignments. Because the MAX-SAT problem of satisfiable instance is equivalent to SAT problem, usually people only study MAX-SAT on unsatisfiable instances. MIN-ONE SAT is not meaningful for unsatisfiable instances since there does not even exist a solution to be optimized. On the contrary MAX-SAT is more meaningful for unsatisfiable instances. In this dissertation, when MAX-SAT is referred, the target instance is assumed as unsatisfiable.

There are various ways to solve the MAX-SAT problem. One of them is based on the SAT local search algorithms [23][73]. In these MAX-SAT local search algorithms, similar to the SAT local search algorithms, certain local greedy heuristics were deployed to search a solution of the instance. When the search ends after a preset limits (time, resource, etc.), usually a near-satisfiable variable assignment can be found. For instance, WALKSAT itself can be used to solve MAX-SAT problems. Although the light-weight and flexibility make local search algorithms suitable in some real applications, generally they lack the capability to assure optimality. From the practicality point of view, the bounds they achieve usually are loose for medium and large instances.

To pursue a guaranteed optimality lots of research turns to extend the DPLL algorithm to solve MAX-SAT problem. Generally they are depth-first branch-and-bound algorithms [7][76][6][18] with a DPLL skeleton. During the search progress, a variable called Upper Bound (UB) records

the current minimal number of clauses that were unsatisfied by an assignment. The number of unsatisfiable clauses by current partial assignment is called $UNSATNUM$. If we have an underestimation UE of how many clauses would be unsatisfiable when the current partial assignment was extended to complete, the best reachable lower bound would be $UNSATNUM + UE$. It is easy to see that when $UB \leq UNSATNUM + UE$ there is no need to search the subspace anymore since no better solution would be found. Clearly this bound estimation helps to prune the search space effectively. When $UB > UNSATNUM + UE$ the branch-and-bound search strategy will continue under this subspace. The algorithm stops when the whole search space has been pruned and searched. There are other ways to solve MAX-SAT problems, like in [17] the author proposed an algorithm based on resolution.

The MAX-SAT problem has some variations. One of them is called Partial MAX-SAT. It is especially useful in scheduling and planning problems [52][51]. In Partial MAX-SAT, a set of clauses is defined as hard constraints which means they have to be satisfied. On the other hand, a set of clauses called soft constraints are expected to be satisfied as many as possible. It is somewhat different from MAX-SAT since in MAX-SAT no clause is forced to be satisfied. Partial MAX-SAT can be solved in a similar manner as MAX-SAT [48][65]. Just as MIN-ONE SAT, in MAX-SAT when considering maximize the satisfiability of the instance, weights on each clause can be different. People use weighted MAX-SAT to refer the problem where some clauses are more favored to be satisfied, but not forced as in Partial MAX-SAT [65][36][8]. One can see that MAX-SAT is a special case of weighted MAX-SAT where each clause is treated equally.

Besides the aforementioned MAX-SAT algorithms, MAX-SAT can be tackled based on MIN-ONE SAT algorithms. The basic technique is that for each clause an auxiliary variable is introduced to help this clause to be satisfied when necessary. Since this set of auxiliary variables is independent to the original formula and irrelevant to each other, this transformed new formula is guaranteed to be satisfiable. Now the MAX-SAT problem of original formula is converted to a MIN-ONE SAT problem on the transformed new formula, where the minimal number of ONES on the auxiliary variables are desired. The underlying principle here is when the auxiliary variable is ZERO the transformed clause is deduced to the corresponding clause in the original formula, thus the MIN-

ONE SAT solution on the auxiliary variables corresponds exactly the MAX-SAT solution of the original formula. It should be pointed out that the MIN-ONE SAT solution on auxiliary variables requires the backbone MIN-ONE SAT solver to be capable to provide MIN-ONE solution on a subset of variables. We know that MIN-ONE SAT can be solved by some generic pseudo Boolean solvers, consequently some pseudo Boolean solvers are also able to solve MAX-SAT problems.

For example, given a CNF formula $(x_1 + x_2)(x_1 + \bar{x}_2)(\bar{x}_1)$. This formula will be transformed to $(x_1 + x_2 + AX_1)(x_1 + \bar{x}_2 + AX_2)(\bar{x}_1 + AX_3)$, with the auxiliary variables as AX_1, AX_2, AX_3 . We call the first formula as F_o and second formula as F_n . The MIN-ONE SAT solution of auxiliary variables in F_n is $\{AX_1 = 0, AX_2 = 0, AX_3 = 1\}$ with associated assignment $\{x_1 = 1, x_2 = 0\}$. Note that the MIN-ONE solution is not unique in this example. It is easy to see the MAX-SAT bound equals to 2 (maximal number of satisfied clauses are two with one clause left unsatisfied). It should be denoted that this MIN-ONE SAT based scheme is very flexible and extendable. For instance the Partial MAX-SAT can be mapped to only adding the auxiliary variables on the soft constraints. Comparatively the weighted MAX-SAT can handled by the weighted MIN-ONE SAT extension.

In Chapter 6, we propose a MAX-SAT solver based on our novel MIN-ONE SAT approximation algorithm. The experiments demonstrated our solver is able to outperform the existing the-state-of-art MAX-SAT solvers over an order of magnitude.

2.5 Preliminaries on the Basic SAT Algorithms

To help the readers to understand the details we will discuss in the following chapters, in this section some preliminary knowledge will be presented.

2.5.1 DPLL and WALKSAT Preliminaries

First We provide more details of the DPLL and the WALKSAT algorithms. 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 2.1 [62].

Listing 2.1: 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 in a given iteration of the algorithm 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 [67] was proposed in 1994 as an improvement of the previous GSAT [66]. First, several definitions will be explained before going deeper in the WALKSAT algorithm, shown in Listing 2.2. 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 2.2: 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 [37]. WALKSAT will try a preset number of steps (CUTOFF) before it claims

that it has failed to find a solution.

2.5.2 MIN-ONE SAT Preliminaries

To help the readers better understand our proposed MIN-ONE SAT approximation algorithm, in this section we will explain the terminology that will be used throughout Chapter 5, followed by the discussion of some of the existing MIN-ONE SAT algorithms.

The target MIN-ONE SAT formula is denoted as F_o . Suppose F_o has N variables, each of which is represented as A_i , $1 \leq i \leq N$. The value of A_i assigned by the MIN-ONE SAT solver is denoted as V_i . As mentioned before, the objective function of basic MIN-ONE SAT is *minimize* $\sum_{i=1}^N V_i$. It can easily be extended to *minimize* $\sum_{i=1}^N c_i \times V_i$, where c_i denotes an integer coefficient corresponding to each variable value.

Listing 2.3: MIN-ONE SAT Based on PBSAT

```

def MIN-ONE_SAT( $F_o$ /*input formula*/)
begin
     $i=0$  /*iteration counter*/
     $BE=$  InitBE();/*return the most pessimistic estimation*/
    while ( )
         $F_c=$ Bound_Transformation( $BE$ );
         $F_A=F_o \cdot F_c$ 
        Result=Call_SAT_Solver( $F_A$ )
        if (Result == UNSATISFIABLE)
            return /*solution found*/
        else
            Update( $BE$ ); /*update based on current solution*/
end

```

We now describe a previous MIN-ONE SAT algorithm based on PBSAT. The complete algorithm is listed in Listing 2.3. To find the optimal solution, the PBSAT algorithm works in an iterative manner. In each iteration, a bound estimate BE is assigned to the objective function, then

the inequality formula ($\sum_{i=1}^N c_i \times V_i \leq BE$) will be transformed into a Boolean formula F_c . There are various techniques to perform the transformation [31], based on the basic conversion of the inequality to a Boolean circuit. The transformation details will not be discussed here since it is out of the scope of this paper. After obtaining F_c , a conventional SAT solver is invoked on the new formula F_A , where $F_A = F_o \cdot F_c$, whose satisfiability indicates whether the bound estimate can be achieved by the objective function. Initially, F_c is set to have a loose bound, usually set at a pessimistic estimate (e.g., set equal to N , the number of variables in the formula), and F_c is gradually tightened by decreasing the value of the summation. If F_A is satisfiable, it indicates that a solution exists that satisfies both F_o and F_c , and the bound estimate, BE , formula F_c , and F_A will be updated accordingly. The algorithm stops when F_A becomes unsatisfiable. The unsatisfiability of F_A guarantees that no better solution exists and the previous bound estimate is the optimal solution.

A different MIN-ONE SAT algorithm has been proposed in [38] which is quite different from the PBSAT technique described above. In [38], the DPLL algorithm is adapted to solve the SAT related optimization problems, such as MAX-SAT[16], MIN-ONE, DISTANCE-SAT[11]. This algorithm is named as Optimization SAT (OPTSAT). The pseudo code of OPTSAT is presented in Listing 2.4.

Like the DPLL algorithm, The OPTSAT performs a decision-based search through a recursive function. The only part that involves the optimization is in the decision heuristic. The OPTSAT maintains a partial order P on all literals, and this order is used to make the next decision in the search. The order in P is defined in a way such that the objective function based on this order is monotonically increased or decreased. For example, given two variables $\{a, b\}$ there exists a P set as $\{\{\bar{a}, \bar{b}\}, \{\bar{a}, b\}, \{a, \bar{b}\}, \{a, b\}\}$. The order of this P set is ascending on the number of ONES in the assignment. One can see that by forcing a partial order on the set of literals the solution space is searched in a specific sequence, where the potential solutions are ordered from the best to the worst. Thus, whenever a solution is obtained, the algorithm returns a solution that is guaranteed to be the optimal solution. OPTSAT achieves good performance in many benchmarks but it also faces the problem that the forced decision order may degrade the solver performance in large formulas. It should also be noted that OPTSAT is not specifically for MIN-ONE SAT; it can solve other SAT

optimization problems like MAX-SAT and DISTANCE-SAT.

Listing 2.4: OPTSAT Algorithm

```
def OPTSAT_DLL(F/*input formula*/,
               A /*Assignment*/,
               P/*literal partial order set*/)
begin
  if (CONFLICT) return FALSE;
  if (F==) return TRUE
  if ( $\{l\} \in P$ )
    return OPTSAT_DLL(assign( $l$ ,F),A $\cup\{l\}$ ,P)
   $l$ =ChooseLiteral(F,A,P);
  Result=OPTSAT_DLL(assign( $l$ ,F),A $\cup\{l\}$ ,P)
  if (Result != FALSE) return Result;
  return OPTSAT_DLL(assign( $\bar{l}$ ,F),A $\cup\{\bar{l}\}$ ,P)
end

def assign( $l$ , F)
begin
  Reduce formula F by  $l$  and its implications
end
```

Besides the above two MIN-ONE SAT algorithms, one may use an all-solutions SAT solver as a base MIN-ONE solver as well. However, because the-state-of-art MIN-ONE SAT solvers are mainly built on the aforementioned algorithms, in this paper we only compare our proposed algorithm with these two algorithms.

2.5.3 MAX-SAT Preliminaries

In this subsection, first we will describe the MIN-ONE SAT based MAX-SAT solver framework, which is shown in Fig 2.3.

The framework can be partitioned into three steps. First the target formula should be trans-

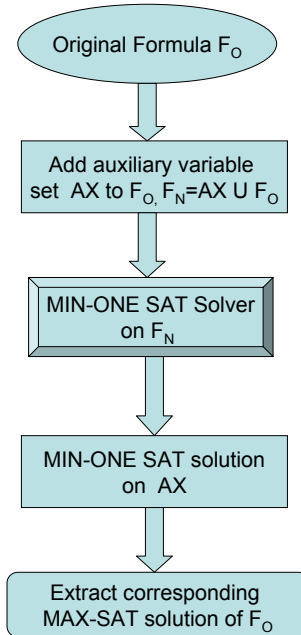


Figure 2.3: MIN-ONE SAT based MAX-SAT Solving

formed into a new formula with auxiliary variables embedded into each clause. Then this new formula is solved by a MIN-ONE SAT algorithm with the capability to find a solution on subset of variables. Finally the MAX-SAT bound and the solution are derived from the MIN-ONE SAT solution of the transformed formula. Since that the transformation and final-step extraction is linear with the size of the clause, the computational complexity is determined by the MIN-ONE SAT solving in the second step.

Listing 2.5: MAXSAT DPLL Algorithm

```

def MAXSAT_DPLL
begin
  while (true)
    if (!decide()) /*if no unassigned vars*/
      if (UNSAT_NUM<UB)
        update UB //the best solution so far
      if (!backtrack()) //find the next solution
        return UB //finished, return optimal bound
  while (!bcp())
  
```

```

        if (!backtrack())
            return UB //finished, return optimal bound
    end
def backtrack()
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

def bcp()
begin
    while (!implication_queue_empty())
        update implication queue
        update UNSAT_NUM
        update UE
        if (UB<=UNSAT_NUM+UE)
            return false
end

```

The DPLL based MAX-SAT algorithm is presented in Listing 2.5. Compared with traditional DPLL algorithm, MAX-SAT DPLL will not backtrack just because a clause is evaluated as false. It records the number of unsatisfied clauses (*UNSAT_NUM*) in the `bcp()` function and keep checking whether the current upper bound *UB* can be exceeded. If not, the `bcp()` function returns false that invokes a backtrack to direct the search into another subspace. When all the variables are assigned, the `decide()` function will return false. At this time *UB* will be updated based on the current *UNSAT_NUM*. The MAX-SAT DPLL algorithm stops when the whole search space is implicitly searched. One can see that the computation complexity of MAX-SAT DPLL is on the

same level of the All-Solution SAT, which is actually NP-hard[49].

Usually both DPLL-based MAX-SAT algorithm and MIN-ONE SAT based MAX-SAT algorithm are complete. It is possible to modify them to search for suboptimal solutions in order to save computational resources. Although the optimality of MAX-SAT is sacrificed in this scenario the performance gain usually can justify the tradeoff.

Chapter 3

Hybrid SAT Solution

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.

In this chapter[32], we present a novel framework that integrates both DPLL algorithm and local search algorithm seamlessly. 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.

This unique hybrid SAT solution will be presented in Section 3.1, followed by experimental results and conclusion in Section 3.3 and Section 3.4.

3.1 Our Approach

We use the WALKSAT v43 algorithm as the base local search part. In the rest of the chapter, the local search solver will be referred to as WALKSAT. The complete HBISAT algorithm is shown in Listing 3.1. 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 $B_i \cup C_{i-1}$ 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 by the DPLL solver can be carried from one iteration to the next [71][75].

Listing 3.1: HBISAT Algorithm

```
def HBISAT_Solver(F/*input formula*/)
begin
    C=0 /*Set of broken clauses*/
    A=0 /*Assignment*/
    i=0 /*iteration counter*/
    CUTOFF=MAX_LOCAL_SEARCH_STEP
    while (true)
```

```

    if (A== $\emptyset$ )
        RandomInitialWalkSAT()
    else
        InitialWalkSAT(A)
    Status=WALKSAT_Solver(CUTOFF)
    if (Status==SAT)
        VerifySolution()
        return SAT
    else if (Status==UNKNOWN)
        Result=CallDPLL(i)
        else if (Result==SAT)
            A=GetAssignmentFromDPLL()
        else
            return Result
    i++ /*count iteration*/
end
def CallDPLL(i)
begin
    B=GetBrokenclause()
    /*guarantee one more clause will be added*/
    B = B  $\cup$  RandomPickOneNewClause()
    C = B  $\cup$  C
    AddClauseToDPLLSolver(C)
    Status=DPLL_Solver()
    return Status
end

```

Theorem 1. HBISAT is complete

Proof. In the HBISAT Algorithm shown in Listing 3.1, 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. \square

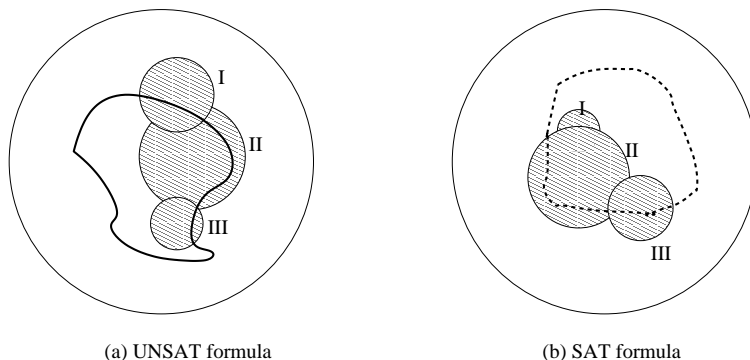


Figure 3.1: HBISAT on SAT & UNSAT formulas

Figure 3.1 illustrates how HBISAT works on SAT formulas. Figure 3.1(a) shows the scenario that the formula is unsatisfiable and Figure 3.1(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 three different broken clause sets generated from three different iterations.

If the original formula is unsatisfiable, there exists at least one unsatisfiable core [80]. 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 the 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 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., $B \cap UC = \emptyset$. This means that WALKSAT must have obtained an assignment that satisfies all clauses outside of B . However, because $B \cap UC = \emptyset$, the clauses outside B contains 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. \square

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.

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 3.1 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.

To better understand HBISAT, it can be explained as an abstraction and refinement scheme, shown in Figure 3.2. Abstraction-refinement is a widely used technique in formal verification [12]. Because abstraction helps to reduce the complexity and refinement guarantees the correctness of the abstract model, abstraction-refinement scheme usually can significantly boost the performance in many applications. Due to fact that the set of clauses that the DPLL solver worked on is a subset of original formula, it can be viewed as an abstraction of the original formula. Thus the partial assignments returned from the DPLL solver actually is a over-approximation of the true

solution. This over-approximation solution could be spurious. 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 DPLL solver to make the abstraction more precise.

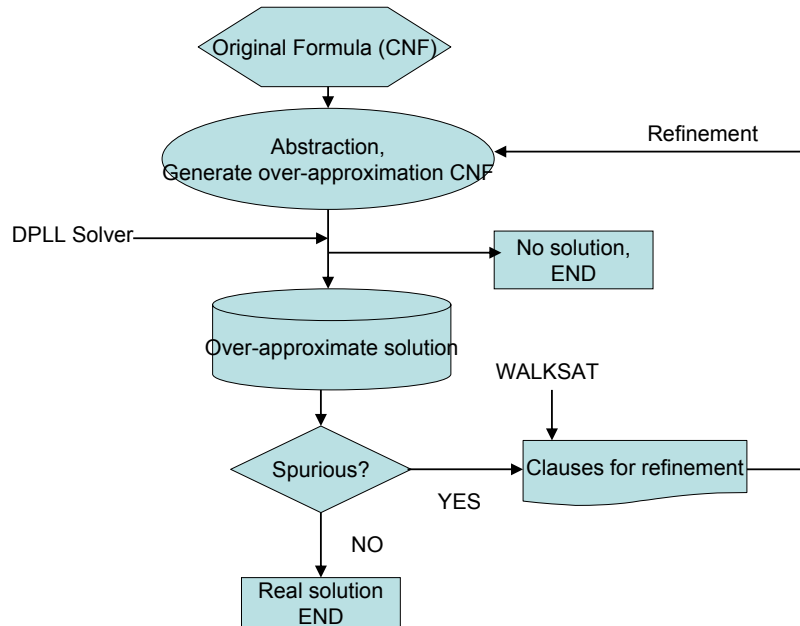


Figure 3.2: Abstraction and refinement in HBISAT

HBISAT actually is a flexible SAT framework. Instead of the specific local search and DPLL solvers we used in this chapter, 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.

3.1.1 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.

3.1.2 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 in a memoryless way 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 original formula plus the conflict clauses generated from

the DPLL solver. Note that this new set of clauses constitutes an expanded formula. It is easy to see that the expanded formula may vary from an iteration to the next. Given the fact the conflict clauses represent the 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. By randomly flipping a will step to a wrong direction because (a) is actually implied by the formula. Suppose an additional conflict clauses $(a + b)$ returned by the DPLL solver was added to this formula, the broken-count would have been updated as $\{2, 1, 2\}$. Based on the broken-count $\{2, 1, 2\}$ WALKSAT flips variable b and leads a new assignment $\{1, 1, 0\}$ with broken-count $\{1, 1, 0\}$. Now in 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 $abc = 111$ is obtained.

When applying WALKSAT on 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 with a bigger weight in broken-count counting, which is 2 in our implementation.

3.2 Study of Synergies

In this section, we will explore the interactions between WALKSAT and DPLL solver through some examples. For Figures 3.3, 3.4, and 3.5, the X-axis denotes the iteration index and the Y-axis denotes the percentage of clauses. In Figure 3.3, an 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

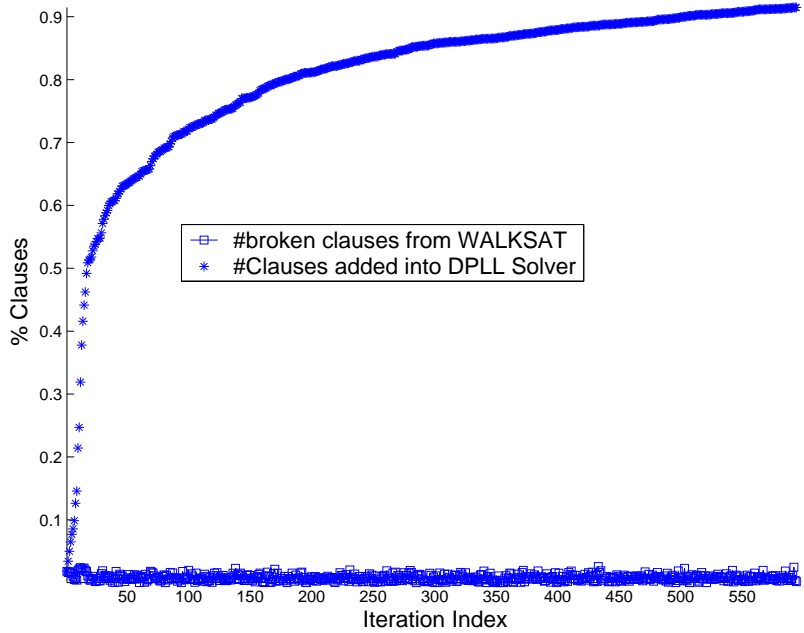


Figure 3.3: Early termination of an UNSAT instance

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 for solving the complete formula.

Figure 3.4 presents an example where 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 satisfiable instance. In contrast, the pure WALKSAT cannot obtain the solution for this instance after 200 iterations with the same CUTOFF value.

If the early termination is not achievable and a solution cannot be found by WALKSAT, the

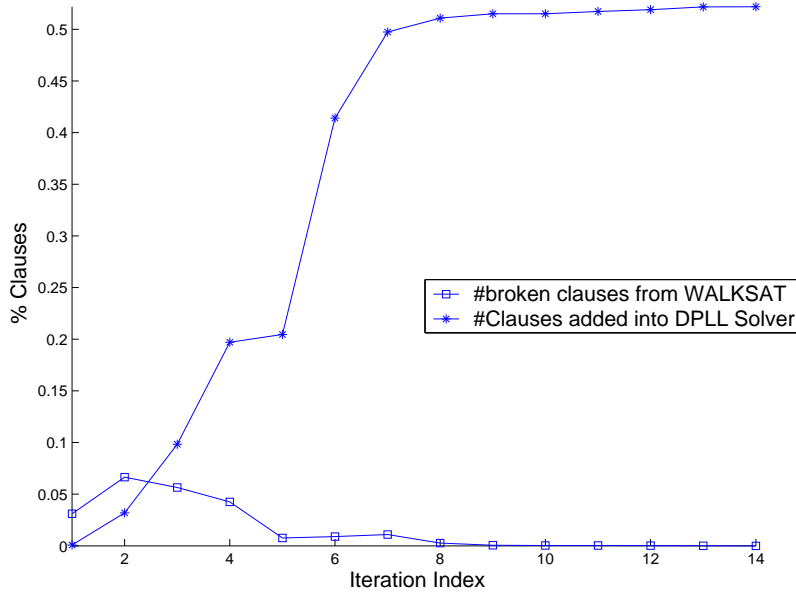


Figure 3.4: Solution found at an early stage by WALKSAT

entire instance will eventually be added into the DPLL solver. Figure 3.5 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.

Figure 3.6 shows an example about the distribution of broken clauses in each iteration. The X-axis is again the iteration index 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. One can see from Figure 3.6 that the broken clauses are clustered. While the size of the broken clauses does not monotonically decrease during HBISAT’s solving process, it can be

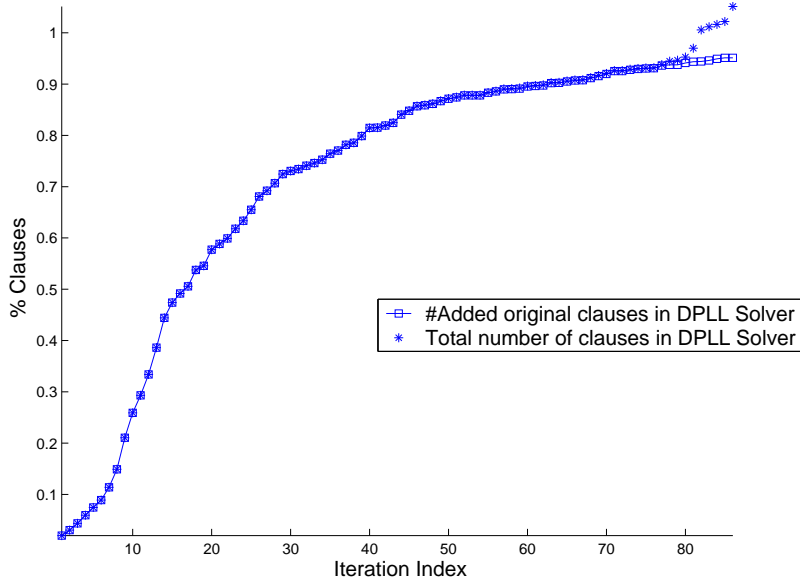


Figure 3.5: Example of gradually learning

observed that at higher iteration numbers, the size of the broken clauses does begin to shrink.

3.3 Experimental Results

The proposed HBISAT was implemented in C++ under the 64-bit Linux operating system, and experiments were conducted on a Intel Xeon 3.0G workstation with 2GB 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 [1]. 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 prob-

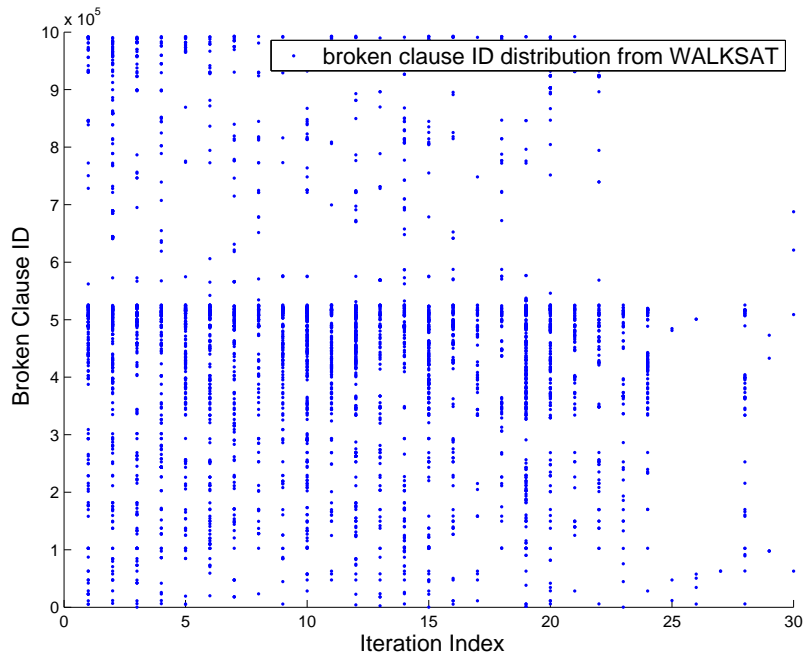


Figure 3.6: Broken clauses distribution returned by WALKSAT

lem [53], 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 [74]. 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 3.1. 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 two of them while HBIZ could solve every instance within 500 seconds, except 06.k60. 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.

Next, experiment II was set up to evaluate the unsatisfiable instances, with the results reported in Table 3.2. The first four columns of Table 3.2 are similar as in Table 3.1. 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. 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 3.3. 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 while HBIm needed only 15.01 seconds. 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.

3.4 Summary

In this chapter, 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.

Table 3.1: Category I Benchmarks

Instance	#V/#C	SAT?	ZChaff(s)	HBIz(s)
01.k40	29249/124460	SAT	99.26	61.73
01.k50	36339/154810	SAT	OUT	177.89
01.k60	43429/185160	SAT	OUT	490.74
03.k40	46225/198298	SAT	31.45	57.56
03.k50	58595/251378	SAT	258.28	195.81
03.k60	70965/304458	SAT	434.55	250.99
06.k40	49126/213666	SAT	124.31	115.54
06.k50	61776/268886	SAT	1921.92	247.1
06.k60	74426/324106	SAT	1709.39	1605.57
07.k40	13151/35904	UNSAT	6.04	20.59
07.k50	15221/40774	UNSAT	5.94	19.19
07.k60	17291/45644	UNSAT	6.06	20.58

Instance	#V/#C	SAT?	Minisat(s)	HBIIm(s)
01.k70	50519/215510	SAT	53.89	53.01
01.k80	57609/245860	SAT	174.8	132.74
01.k90	64699/276210	SAT	330.41	60.03
03.k70	83335/357538	SAT	45.84	76.39
03.k80	95705/410618	SAT	81.71	80.19
03.k90	108075/463698	SAT	267.64	197.57
06.k70	87076/379326	SAT	60.22	109.09
06.k80	99726/434546	SAT	201.19	135.64
06.k90	112376/489766	SAT	296.07	189.53
07.k70	19361/50514	UNSAT	9.86	15.33
07.k80	21431/55384	UNSAT	1065.7	17.45
07.k90	23501/60254	UNSAT	5.97	16.21

Table 3.2: Category II UNSAT Benchmarks

Instance	#V/#C	ZChaff	HBIz	Minisat	HBIIm
c7b	26058/77128	147.4	149.3	55.54	54.27
c8n	53697/159595	386.93	363.89	146.18	169.25
c9b	36757/109045	407.53	381.7	263.54	226.02
f9n	185149/552412	2011	1604.91	OUT	OUT
g9n	54631/161950	251.33	222.53	61.84	84.12
g9b	59110/175387	208.65	233.03	41.49	106.8
g9idw	125885/371998	250.55	263.39	70.64	136.11
g9nidw	170918/506584	1552.48	1098.25	942.97	691.3
c10	17121/50803	9.87	16.25	12.15	24.44
c10b	43517/129265	669.88	529.1	589.1	567.78
c10bi	147116/437224	OUT	OUT	OUT	OUT
c10bid	291912/828039	OUT	OUT	OUT	OUT
Total		5476	4549.4	2183.45	2060.09

Table 3.3: Category III Benchmarks

Instances	#V/#C	SAT?	ZChaff(s)	HBIZ(s)	Minisat(s)	HBIIm(s)
fvp-sat3.0/pipe-64-4-bug01.cnf	35853/1021170	SAT	40.83	8.17	1.55	4.29
fvp-sat3.0/pipe-64-4-bug02.cnf	35853/1021171	SAT	OUT	14.81	OUT	OUT
fvp-sat3.0/pipe-64-4-bug03.cnf	35947/992674	SAT	4.8	11.19	0.97	4.28
fvp-sat3.0/pipe-64-4-bug04.cnf	35854/1012315	SAT	39.07	11.16	4.39	7.84
fvp-sat3.0/pipe-64-4-bug05.cnf	35853/1022271	SAT	555.54	40.86	OUT	OUT
fvp-sat3.0/pipe-64-4-bug06.cnf	35853/1022271	SAT	282.08	6.67	OUT	9.4
fvp-sat3.0/pipe-64-4-bug07.cnf	35853/1022271	SAT	997.67	28.41	OUT	15.01
fvp-sat3.0/pipe-64-4-bug08.cnf	35622/1003074	SAT	38.5	6.44	OUT	4.63
fvp-sat3.0/pipe-64-4-bug09.cnf	35726/1011764	SAT	0.4	10.09	180.03	5.29
fvp-sat3.0/pipe-64-4-bug10.cnf	35839/1012135	SAT	0.62	12.18	1242.21	3.6
fvp-sat3.0/pipe-64-4-bug11.cnf	35853/1012271	SAT	1148.21	93.65	2.6	10.37
Total			3107.73	243.63	1431.75	64.71
fvp-unsat2.0/5pipe.cnf	9471/195452	UNSAT	10.99	30.78	78.77	185.66
fvp-unsat2.0/5pipe-5-ooo.cnf	10113/240892	UNSAT	34.30	52.75	OUT	OUT
fvp-unsat2.0/6pipe.cnf	15800/394739	UNSAT	86.98	66.52	OUT	OUT
fvp-unsat2.0/6pipe-6-ooo.cnf	17064/545612	UNSAT	159.35	276.63	252.53	205.35
fvp-unsat2.0/7pipe.cnf	23910/751118	UNSAT	275.58	284.23	OUT	OUT
fvp-unsat2.0/7pipe-7-ooo.cnf	24415/711050	UNSAT	2768.58	2425.08	OUT	OUT

Chapter 4

Double-Layer Conflict Driven Learning with Pseudo Boolean Constraints

As one of the most crucial techniques in SAT solver, conflict-driven learning remains a challenging topic to be improved or refined. Conflict-driven learning helps to prune the search space by intelligently learning from conflicts in its search history. Due to its aggressive learning strategy, the effectiveness of conflict-driven learning may be impaired by an overwhelmingly large number of clauses learned that may turn out to be useless information. Based on our observations, we found that a carefully designed selective technique can benefit conflict-driven learning tremendously, where the conflict-induced clauses are classified as primary or supplementary. Furthermore, to incorporate with two different categories of conflict-induced clauses, pseudo Boolean constraints are introduced to more compactly store the supplementary conflict-induced clauses while keeping a portion of the learned knowledge for search space pruning.

In this chapter, we first explain some details about the pseudo Boolean constraint set. Then we will introduce how our double-layer conflict driven learning works in Section 4.1, followed by the in-depth analysis of conflict-induced clause evaluation in Section 4.2. The experimental results are presented finally in Section 4.3.

4.1 Two-layer Conflict Driven Learning

4.1.1 Pseudo Boolean Constraint Set

We first provide a quick review on Pseudo Boolean Constraints (a.k.a. Pseudo Boolean Clauses). A pseudo Boolean constraint is sometimes also referred to as a linear pseudo Boolean constraint, which is usually represented as $\sum a_i \cdot x_i \geq b$ where a_i and b are integers and each of the x_i is a Boolean variable. One can see that when $b = 1$ and $a_i = 1$, the pseudo Boolean constraint simply becomes a Boolean clause. Due to its expressive power, it is possible to represent a set of complex Boolean clauses with a much fewer number of pseudo Boolean constraints. For example, consider four Boolean clauses $(a + b + \bar{c})(a + \bar{b} + c)(\bar{a} + b + c)(a + b + c)$. They can be expressed with just one pseudo Boolean constraint $(a + b + c) \geq 2$. It should be pointed out that given a set of Boolean clauses it could be extremely difficult to find its equivalent pseudo Boolean constraint.

In our proposed double-layer approach, PBCS is a small set of pseudo Boolean constraints. When a CIC is assigned to be stored in PBCS, we will first select the most appropriate PBC to which the CIC will be added. Then, the CIC is translated to a pseudo Boolean constraint and added to the selected PBC: $PBC_k = PBC_{k-1} + PBC_{CIC}$. For instance, if the selected PBC is $2x_1 - x_2 + x_3 \geq -1$, with the CIC as $(x_1 + x_3)$, the translated CIC is $x_1 + x_3 \geq 1$ and the updated PBC would be $3x_1 - x_2 + 2x_3 \geq 0$. Note that a negative literal \bar{x} is translated as $1 - x$.

One can easily observe that at each summation the blocking power of a conflict-induced clause may not be completely kept. In our example, the clause $(x_1 + x_3)$ blocks the space of $x_0 = 0, x_3 = 0$. In the new PBC $3x_1 - x_2 + 2x_3 \geq 0$ only $x_0 = 0, x_2 = 1, x_3 = 0$ would be blocked. Generally, the blocking power from CIC can be kept, weakened, or completely lost, depending on how and where it is added. However, we would like to retain as much power from the supplementary clauses as possible. We follow up this discussion with an example. Let the PBC be $x_1 + x_2 \geq 1$, and the CIC be $(x_1 + \bar{x}_2)$. By adding the CIC to the PBC, the resulting PBC becomes $(x_1 + x_2 \geq 1) + (x_1 - x_2 \geq -1) \Rightarrow x_1 \geq 1$. It can be seen that this is exactly the same as binary resolution, and in this case no blocking power will be lost. It is quite apparent that when more cases are

analyzed, we find that more common variables between PBC and CIC translate to blocking power retainment in the summation operation. We use this observation as the heuristic to pick PBC in PBCS to accommodate the current CIC. In our implementation, the PBCS is created with ten PBCs. Each PBC covers a portion of variables in an ascending order of the variable ID. A CIC will be inserted to the PBC with the maximal variable overlap based on our variable range partition.

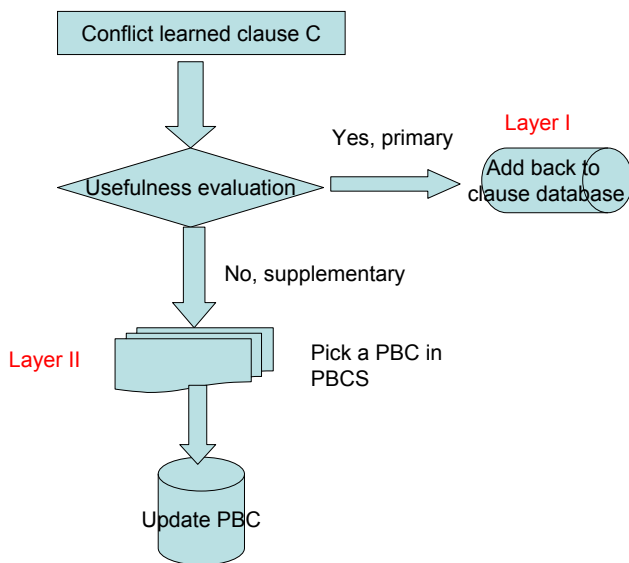


Figure 4.1: Double-layer conflict driven learning algorithm

4.1.2 Basic Algorithm

The basic algorithm of our double-layer conflict-driven learning is shown in Figure 4.1. Different from traditional SAT solvers, whenever a conflict-induced clause is obtained, there is an extra step to evaluate whether the CIC is primary or supplementary. Only those primary ones will be added back to the clause database. All the supplementary clauses are assigned to the PBCS to update the PBCs.

In order to utilize the blocking power of PBCS, both the conflict checking procedure and BCP

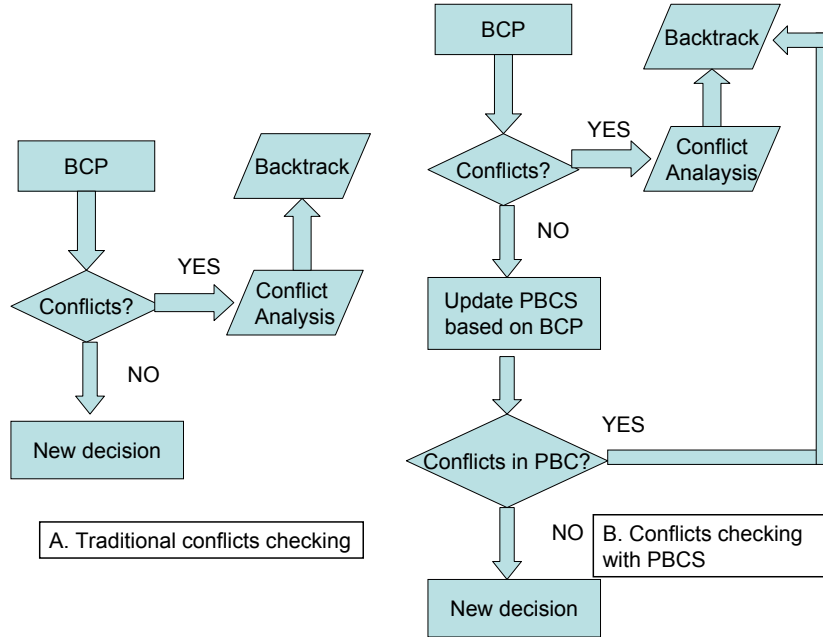


Figure 4.2: Conflicts checking traditional vs. with PBCS

procedure are also updated, as shown in Figure 4.2. Figure 4.2 compares the difference between the traditional SAT solver and ours. Note that now we must check for conflicts in both the Boolean clause database and PBCS. Whenever a conflict occurs in the Boolean clause database, conventional conflict handling mechanism is invoked. When no conflict occurs in the Boolean clause database, but at least one pseudo Boolean constraints is violated, the SAT solver will also backtrack immediately. However, no conflict clause will be produced. In case conflicts occur in both the clause database and PBCS, we put higher priority on the Boolean clause because it provides another opportunity to learn a new conflict clause. Accordingly the BCP procedure is also modified. Each PBC will be updated and checked at each iteration of variable implication.

We want to make it clear that backtracking from conflicts in PBCS is sound and will not change the completeness of SAT solver. Theorem 3 is formulated to prove this fact.

Theorem 3. Given a pseudo Boolean constraint C to be the summation of a set of Boolean clauses, $C = \sum_{i=1}^{i=n} S_i$. If there exists an assignment A that violates C , then under A at least one of the Boolean clauses will be evaluated as false.

Proof. We prove this by contradiction. Since every Boolean clause S_i is satisfied under the assignment A , we know $\forall i, S_i > 0$. Therefore, $C = \sum_{i=1}^{i=n} S_i > 0$. However, assignment A violates C , $C \leq 0$. Thus, a contradiction. \square

Based on Theorem 3, a conflict in PBCS corresponds to a conflict on at least one of the supplementary conflict-induced clauses. Thus, it is sound to backtrack when conflicts occur in PBCS. Note that the converse is not true, i.e., a conflict in a supplementary clause may not always cause the PBCS to have a conflict. Therefore, the proposed PBCS retains a portion of the blocking power of supplementary CICs.

4.1.3 Recovery Hash Table

Since SAT solving is highly dynamic and can be very sensitive to many factors, it is almost impractical to determine the usefulness of a conflict-induced clause at the time of its birth. Imagine what would happen if a clause is dropped simply because it is erroneously determined to be supplementary, but actually it is very important to the problem instance. This indicates that we need a mechanism to remedy the damage when the approach mis-predicts the usefulness of a conflict-induced clause. Our solution comes from an assumption that if one clause is learned repeatedly, this conflict clause should be treated as primary. The underlying explanation here is that a frequently met conflict clause can help to block the portion of the search space if it was added to the clause database at the beginning.

We build a hash table to record the conflict-induced clause occurrence. This hash table is called Recovery Hash Table (RHT). All the entries of RHT are initialized to 0. For each CIC, before the evaluation a hash value (signature) G is computed and the corresponding position in the hash table ($RHT[G]$) will be checked. If $RHT[G] == 1$, which means this CIC most likely has been computed before, we will categorize this CIC as primary and reset the hash table entry. If not, CIC will be passed to the evaluation function. When the evaluation function identifies this CIC as supplementary, $RHT[G]$ will be set to 1. This approach essentially prevents adding an

important CIC to the PBCS repeatedly. Instead, it ensures any repeated generated CIC is treated as primary and will be immediately added to the clause database. The recovery hash table is simple to implement, which only adds a very small runtime and memory overhead. The entire flow is illustrated in Figure 4.3.

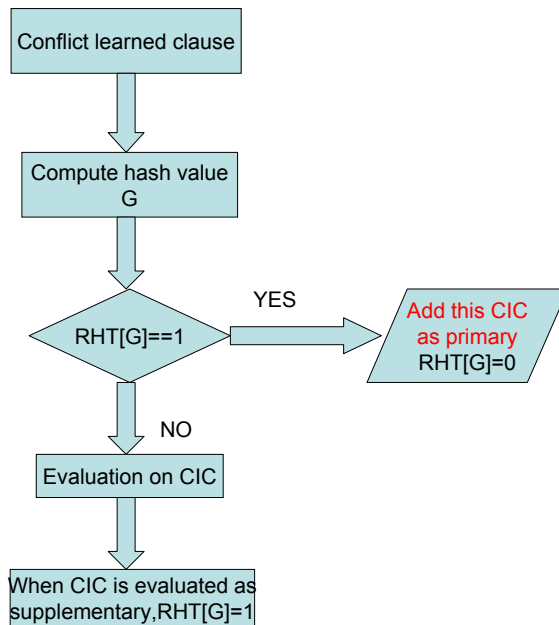


Figure 4.3: Recovery hash table working flow

4.2 Evaluation of Conflict Induced Clause Usefulness Estimation

4.2.1 Evaluation Metrics

Evaluating the usefulness of a conflict-induced clause can be extremely difficult. We start our study through carefully analyzing the factors that have significant impact on the usefulness of CIC. Let us first consider a typical scenario. Assume a CIC c was learned and it was dropped. In the subsequent SAT solving process, at some point suppose c is violated explicitly by the current variable assignments. We have to go through the whole implication process again since c is not currently in the clause database. We can see that compared with the scenario where c was never

dropped, the unnecessary cost would be the cost of traversing and analyzing the implication graph. The cost of this conflict analysis is directly proportional to the size of implication graph. In other words, one metric for the cost related to CIC is the size of the implication graph.

Beside the size of the implication graph, the backtrack level from the CIC c is also very crucial. Due to the non-chronological backtracking technique[45], it is often that the SAT solver backtracks more than one level from its current level. Usually the backtrack level interval roughly determines the blocking power of this CIC. This can be explained in the following manner. All the decisions in the decision stack between the current level and the target backtrack level constitute a set of variables. We know that the current assignments on a subset of these variables led to a conflict. During the future search, other assignments on these variables may also induce conflict on c since structurally these variables are correlated through a set of clauses. The bigger size of the set of variables, more assignment combinations are likely to be blocked by c . Therefore, backtrack depth is defined as our second metric for determining the usefulness of a CIC.

Thus far, the discussion has focused on the loss of dropping a CIC and the potential gain of deductive power of a CIC. Another issue that we need consider is the derivation of new CICs from a previously learned CIC. For instance, a CIC that can be obtained from a subset of the original clause is considered more trivial than another CIC derived that need the knowledge from the first CIC. In other words, the power of a CIC can be carried over many generations and produce powerful CICs when the search goes on. To evaluate the carry-over effect of a CIC, we introduce a new concept called learning level. Each conflict-induced clause carries a learning level, which is defined as 1 level larger than the largest learning level of its literals. A literal's learning level is simply its antecedent's learning level. The antecedent here is defined as the clause from which the literal was implied. So the set of antecedents from a CIC's literals represents the reason why this CIC is implied. Thus, the learning level indirectly measures the number of past CICs the newest CIC is derived from.

Note that all the original clauses in the formula are set at learning level 0. Thus, every CIC has a learning level of at least 1. For instance, if clause c_1 is learned and each of its literal is implied

by a original clause in the formula, then the learning level of c_1 is 1. When c_1 is involved to reason another conflict learned clause c_2 , c_2 has a learning level of 2, and so on.

So far, we have introduced three evaluation metrics: size of implication graph, backtrack depth, and learning level. While these metrics are considerably effective, sometimes certain randomness in the solving process is needed. We bring up randomness as forth metric to balance the situation where the aforementioned metrics do not perform well. Randomness has been widely used in local algorithms to skip over local optima. It also has a long history in SAT solvers. For all the supplementary CICs, we can preset a percentage of randomness to mark them as primary. This value can be adjusted depending on the level of randomness.

4.2.2 Evaluation Function

Due to the complexity of SAT problems, it is rare that one of these four metrics alone is sufficient to determine the usefulness of a CIC. Therefore, we need to combine them somehow. A evaluation function was built based on the four metrics to evaluate the CIC. The evaluation function is expressed as a Boolean function of a set of criteria. Any CIC for which the evaluation function returns *true* will be categorized as supplementary, otherwise it is regarded as primary. Here is an example of the evaluation function: $\Phi = \{(SI < 20) \cdot (BD < 10) \cdot (LL > 2, LL < 5)\}$, where SI denotes the size of implication graph, BD represents the backtrack depth, LL is the learning level, and \cdot is the AND logical operator. This example evaluation function marks any CICs as supplementary if its size of implication tree less than 20, *and* its backtrack depth less than 10 and learning levels between 2 and 5. The randomness factor R is not counted in this example evaluation function. It is also possible to build an evaluation function by OR logic, such as $\Phi = \{(SI < 10)|(BD < 10)|(LL < 3)|(R == 0.1)\}$, where the metric of randomness is introduced. More specifically in this example, with a probability of 10% a CIC will be marked as supplementary. The annotations used here will also be carried to the next section when we describe the evaluation function under different experimental setups.

One may raise the concern that the evaluation function could be quite sensitive to the underlying application, thus there may not exist a universal evaluation function to benefit all the applications from different domains. We agree that finding a good evaluation function is not easy. In our current work, we calibrated our evaluation function based on the general rules that we will discuss later. Considering the vast difference between different application domains, it is reasonable to think that evaluation function can be specifically tuned when extra knowledge is known, such as the structure of the circuit. It is very easy to alter the evaluation function in our proposed architecture, which makes this technique easily adopted to many application domains where the problems are highly structured or weighted heavily on some properties. For instance, when dealing with design verification instances from bounded model checking, those CICs that be linked with the property more closely should be favored in the evaluation function. For those grouped problems sharing a similar structure, such as instances from the pigeon-hole suite, it is helpful to design a evaluation function to tackle the hard ones by sampling the easy ones. With such additional information from the problem instance, adding other metrics may also be useful. In this chapter, we only discuss the four general metrics of SAT formulas without any assumption on the problem’s domain knowledge.

Beyond the flexibility that the evaluation function can provide, there are basic rules that are in common. For example, a CIC with a larger backtrack depth usually should tend to be evaluated as primary. These basic rules of each metric are summarized in Table 4.1.

Table 4.1: Rules of Usefulness Metrics

Metric	Symbol	Primary?	Rule
Randomness	R	↑	Uncertain
Implication graph size	SI	↑	↑
Backtrack depth	BD	↑	↑
Learning level	LL	↑	↑ but not always, mid-level can be more important

In Table 4.1 the first and second columns show the metric and its symbol. The third column shows the change of the value of each metric that corresponds to greater likelihood that the CIC will be determined as a primary clause. The last column shows the rules. These are very primitive

rules, but they set up a solid base of how to put them together to create a good evaluation function.

4.3 Experimental Results

Our proposed double-layer conflict-driven learning was implemented on top of a popular SAT solver, ZChaff version 2007.3.12. The whole program was written in C++ and run under Linux on a workstation with Intel Xeon 3.2G CPU and 2G memory. We name our implementation as DLSAT.

Since the evaluation function is a crucial part in DLSAT, we first set up a case study to analyze the impact of different evaluation metrics, shown in Table 4.2. The first column of Table 4.2 shows the instance name, which is from SAT 2002 competition benchmark[2]. The second column presents the evaluation functions based on each metric. There are total 7 configurations of evaluation function. The results of each evaluation metric are presented according to the evaluation function. Note that “NOSUPP” indicates the evaluation function always returns false, which means every CIC will be treated as primary (no supplementary clauses). It is easy to see that for the “NOSUPP” evaluation DLSAT has the identical results as ZChaff. The following 5 columns are statistics returned by the SAT solver, namely the maximum number of decision levels, the number of total decisions, the number of CICs added to the clause database (primary), the number of literals of the primary CICs, the number of conflicts generated by PBCS, the number of supplementary CICs, the number of hits in recovery hash table, and finally the run time. For example, $(BD < 5)$ means that all CICs whose backtrack depth is less than 5 will be determined as supplementary. In this case, we saw the loss of dropping those CICs with high backtrack depths.

From Table 4.2, we can see that random dropping ($R = 50\%$) performs poorly. This is reasonable because random dropping may accidentally drop the critical (primary) clauses that help to solve the formula. For this specific instance $BD < 5$ slows down the SAT solver while $LL < 3$ improves the performance greatly. The performance is improved largely due to the better decision is obtained when supplementary CICs are filter out. This can be confirmed by comparing the number

of decisions between “NOSUPP” and $LL < 3$. With almost the same maximum decision level, the number of decisions is reduced to half with $LL < 3$.

We already saw that the learning level has the huge impact on the outcome. When LL is set as less than 3, the SAT solver speeds up drastically. The band-pass filter strategy gives even better results. The worst case comes from filtering out high learning level CICs. This is intuitive as CICs with greater learning level are much more important than lower learning level CICs due to reason that they usually contain more inductive power. The reason that band-pass strategy can be better is that filtering too many low learning level CICs hurts the potential to generate powerful high learning level CICs. In other words, we need a base of low learning level CICs in order to learn those more advanced CICs.

There is no universal rule to set the parameter on BD , SI and LL . Actually each metric has its strength and weakness. A good evaluation function can aggregate the effectiveness of each metric to bring up the largest potential performance gain.

A set of SAT instances from various publicly available benchmarks were selected in our study, such as SAT competition 2002, parameterized benchmark suite of Hard-Pipelined-Machine verification problem, formal verification instances of buggy processor and IBM formal verification library [2][1][74][53], named from C1 to C20. The detailed instance list is presented in the Table 4.5, where the complete CNF name and corresponding notion are listed in the first and second column. Since these instances are from broad application domains, we use them to test the viability of DLSAT. The results are reported in Table 4.3. In Table 4.3, the first column reports the instance. If both ZChaff and DLSAT finished within the preset time out limit (1000 seconds), column 2 will report “SAT” or “UNSAT”. “ABORT” means both of them timed out. If one of them aborted, the outcome from ZChaff and DLSAT are shown separated by a slash. Columns 3 to 5 report the statistics of ZChaff, including run time, the number of CICs and the corresponding number of literals. Similarly, columns 6-12 report the results from DLSAT. In addition to run time, primary CICs and their number of literals, four more are reported. They include the number of conflicts detected in PBCS, the number of supplementary CICs, the number of literals of supplementary CICs, and the

number of hits in remedy hash table (RHT). Finally, the speedup of DLSAT over ZChaff and the storage reduction ratio (SR) are reported. The speedup is calculated as $\frac{ZTIME}{PTIME}$. SR is proposed to evaluate how much storage can be reduced when many supplementary CICs are compressed in PBCS. It should be noted that in PBCS each literal costs two storage units. One is for the variable index and the other is for its coefficient as pseudo Boolean constraint allows coefficient on each literal. Therefore we use the following formula to compute SR: $SR = 1 - \frac{2*\#PBCLit + \#PCLSLit}{\#ZCLSLit}$. When either ZChaff or DLSAT aborted after 1000 seconds, statistics of the SAT solver at that moment are snapped and reported. The average of speedup and SR are shown at the last row of Table 4.3.

Among 20 instances, ZChaff aborted on 3 of them while DLSAT aborted 2. Except for the two aborted instances, DLSAT outperforms ZChaff on 11 instances in those 18 finished instances. The average speedup of DLSAT is 3.76 excluding C8, in which a speedup of more than 912 \times was achieved, which is almost three orders of magnitude. In C8, a better decision order can be obtained by compressing the supplementary CICs to reduce the noise introduced by those less-useful CICs. In C7, C17, and C19, more than an order of magnitude speedup was achieved. The slow-downs generally are for those easy instances, where the overhead of DLSAT increased the run-times, such as in C1, C10, and C16. Besides the performance improvement, DLSAT saves storage resource by 22.45%. Over 50% memory consumption is reduced for 7 instances out of 20. Even for the unfinished instance C3, after 1000 seconds running DLSAT saved over 20% on memory. The evaluation function used in this experiment is shown below the table, which is a function of metric SI , BD and LL . In this evaluation function, 1) BD is not fixed. It is determined by the size of the current decision stack, with a limit of less than 30; 2) SI is defined as in proportional to BD with a factor of 5; 3) LL is a band-pass filter with thresholds at 3 and 30. Due to the importance of learning level, any CICs falls outside the band-pass filter will be marked as primary immediately. Otherwise, they will be screened by another two metrics, BD and SI .

In the next experiment, a group of instances from [74] was tested to validate our hypothesis that instances sharing common structures can benefit from the same evaluation function. These instances are generated from the formal verification of buggy invariants of an out-of-order super-scalar processor from CMU. The results are presented in Table 4.4, which has the exact layout as

Table 4.3. Note that those instances that can be solved by ZChaff in less than 1 second were not included in the Table. From Table 4.4 we can see that nearly 6 times speedup and 50% storage reduction are achieved for these 16 instances. We want to emphasize that this experiment reveals a very practical aspect about the evaluation function: it has been observed that instances from a specific domain or problem share significant similarities between them. Thus it is highly possible to customize the evaluation function to pump out more potential on performance improvement and memory reduction. This was supported by our experimental results. As pointed out earlier, one can sample and test the best evaluation strategies on easy instances first to help solving the hard instances later.

Although we did not report the results on different evaluation functions, our results have already demonstrated that under the double-layer conflict learning framework the DLSAT is able to improve performance significantly, while simultaneously saving the memory usage greatly. This can be even amplified by taking advantage of structural similarities sharing between a group of instances, as we saw in Table 4.4.

4.4 Summary

We have presented a novel double-layer conflict-driven learning technique in this chapter. This method can be easily ported to any DPLL based SAT solver with slight modification. Besides giving the basic algorithms, we also analyzed the usefulness of conflict-induced clause. Four metrics to evaluate the conflict-induced clauses are discussed. Evaluation functions based these metrics are tested in the experiments. The experimental results confirm the value of the proposed technique on both performance improvement and memory reduction.

In the future, more research will be investigated on the evaluation function. One possible direction is to exploit the feasibility of introducing a dynamic, self-learning mechanism to adjust the evaluation function on fly.

Table 4.2: Importance of Individual Evaluation Metrics

CNF	Evaluation	max dlevel	# decisions	Added CL	Added Literals	#Conflicts	#CL in PBC	Hash hits	Run time(s)
ezfact48_1.cnf	NOSUPP	82	61900	45682	6593215	0	0	0	31.88
	$R = 50\%$	91	138164	68587	10649672	2871	40830	27603	80.79
	$SI < 10$	128	119255	66802	9706565	1172	42	34	54.1
	$BD < 5$	145	303520	94520	17242656	1438	126458	93728	151.19
	$LL < 3$	106	50182	31392	3933086	805	1732	1496	24.29
	$3 < LL < 10$	89	43013	26343	3477846	1124	2288	1539	21.03
	$LL > 10$	82	389022	153646	30117340	1230	185277	145355	276.54

Table 4.3: General SAT Instances Results

CNF	Result	ZCHAFF			DLSAT							Speedup	SR
		ZTIME	#ZCLS	#ZCLSLit	PTIME	#PCLS	#PCLSLit	#CF	#PBC	#PBCLit	#Hits		
C1	UNSAT	10.59	13263	939267	11.21	10222	773985	11666	2092	12987	1447	0.94	14.83%
C2	UNSAT	17.83	55996	2090004	15.02	47836	1958266	10052	352	1359	317	1.19	6.17%
C3	ABORT	1000.00	235100	12459272	1000.00	191334	9369492	7865	76905	75855	57239	1.00	23.58%
C4	UNSAT	86.30	43640	13777557	103.92	37534	11133036	5154	1018	15560	685	0.83	18.97%
C5	UNSAT	5.60	27102	713189	1.10	7733	172222	6833	22	224	15	5.08	75.79%
C6	ABORT	1000.01	1084667	61819438	1000.01	1049344	61229825	6912	5	185	5	1.00	0.95%
C7	SAT	34.98	36274	716766	1.80	2151	30037	1115	25	310	12	19.47	95.72%
C8	ABORT/SAT	1000.00	535155	83497012	1.10	2837	178238	7494	1212	8000	644	>912.55	99.77%
C9	SAT	36.03	15435	512301	29.67	8516	280330	8246	1671	4186	1132	1.21	43.65%
C10	UNSAT	19.60	77808	829788	32.75	106948	1352270	8448	84	348	84	0.60	-63.05%
C11	SAT	25.22	116877	987154	32.71	118128	992854	8669	18493	19724	15400	0.77	-4.57%
C12	UNSAT	266.47	603517	7918246	263.78	592182	7743311	7846	61559	37411	57725	1.01	1.26%
C13	UNSAT	69.26	163489	2568645	443.94	382701	6534969	6853	45292	2899	45253	0.16	-154.64%
C14	UNSAT	9.78	105994	1387922	4.83	61249	692182	8685	1142	2342	1099	2.03	49.79%
C15	SAT	98.56	67320	1683221	258.98	101051	2715502	2234	26932	71329	18819	0.38	-69.80%
C16	SAT	3.08	4118	253714	6.55	5218	326369	10676	769	14355	577	0.47	-39.95%
C17	SAT	5.22	7815	1978822	0.27	57	11158	6945	1	104	0	19.21	99.43%
C18	UNSAT	4.54	9579	837214	3.62	6242	465273	7971	25	1233	117	1.25	44.13%
C19	SAT	88.47	80288	16598146	8.09	9430	995731	3019	3518	23021	1715	10.94	93.72%
C20	UNSAT	31.36	45682	6593215	13.90	18579	2497441	13291	167	2837	90	2.26	62.03%
Average												3.67*	19.89%

!Evaluation Function $\Phi = \{(3 < LL < 10) \cdot ((SI < 10 * BD) | (BD < Depth))\}$, Depth= min(current_decisionstack_size/8, 30).

*C8 was not included in the average computation

Table 4.4: Experiments on Group of Instances That Shares Similarities

CNF	Result	ZCHAFF			DLSAT							Speedup	SR
		ZTIME	#CLS	#CLSLit	PTIME	#CLS	#CLSLit	#CF	#PBC	#PBCLit	#Hits		
bug01.cnf	SAT	35.09	15435	512301	10.06	2442	86824	9436	802	1797	558	3.49	82.35%
bug02.cnf	ABORT	1000.00	208211	10681078	1000.01	164469	7907126	8778	65208	61211	51154	1.00	24.82%
bug03.cnf	SAT	4.16	1190	33606	4.82	668	26481	6961	128	406	96	0.86	18.79%
bug04.cnf	SAT	33.73	11849	405192	55.24	12977	479461	9183	2553	5466	1700	0.61	-21.03%
bug05.cnf	SAT	469.02	146603	6649645	204.13	43153	1816111	9440	18506	17354	12800	2.30	72.17%
bug06.cnf	SAT	241.65	68078	2819472	107.05	25833	1012356	9284	8538	11821	5825	2.26	63.26%
bug07.cnf	SAT	875.26	241813	11240477	632.42	112196	5212136	9299	47153	44444	35788	1.38	52.84%
bug08.cnf	SAT	33.53	13193	420202	0.48	50	1363	1	1	132	0	70.44	99.61%
bug11.cnf	ABORT/SAT	1000.00	238943	11714369	536.17	88946	4050793	9136	39012	40184	28763	>1.87	64.73%
bug13.cnf	SAT	33.81	10699	348109	14.75	2738	93484	9160	289	1275	193	2.29	72.41%
bug15.cnf	SAT	587.75	142226	6876384	572.05	105994	4876930	9612	53829	41076	40802	1.03	27.88%
bug16.cnf	SAT	313.30	89673	3832856	148.77	30758	1225178	10152	4977	10201	3291	2.11	67.50%
bug17.cnf	SAT	572.68	147968	6688951	500.03	87445	4015001	9562	31708	36649	23216	1.15	38.88%
bug18.cnf	SAT	317.13	99315	4480948	362.60	76770	3446783	9258	31794	27419	23288	0.87	21.86%
bug19.cnf	SAT	170.84	52916	2074267	173.49	39746	1618980	9205	19385	17403	13575	0.98	20.27%
bug20.cnf	SAT	393.67	112968	5081804	268.43	51920	2189750	9223	22645	25681	15904	1.47	55.90%
Average												5.88	47.64%

*Evaluation Function $\Phi = \{(3 < LL < 30) \cdot ((SI < 10 * BD) || (BD < Depth))\}$, Depth= min(current_decisionstack_size/8, 30).

Table 4.5: Instance List

C1	5pipe.cnf
C2	avg-checker-4-23.shuffled.cnf
C3	comb1.shuffled.cnf
C4	f2clk_30.shuffled.cnf
C5	homer06.shuffled.cnf
C6	homer20.shuffled.cnf
C7	IBM_FV_2004_rule_batch_01_SAT_dat.k30.cnf
C8	lisa20_1_a.shuffled.cnf
C9	pipe_64_4_bug01.cnf
C10	s61-100.cnf
C11	sat-grid-pbl-0070.sat05-1334.reshuffled-07.cnf
C12	unsat-grid-pbl-0080.sat05-1344.reshuffled-07.cnf
C13	Urquhart-s3-b10.shuffled.cnf
C14	grid_20_20.shuffled.cnf
C15	IBM_FV_2004_rule_batch_01_SAT_dat.k40.cnf
C16	3bitadd_31.shuffled.cnf
C17	med17.shuffled.cnf
C18	qg7-12.shuffled.cnf
C19	pyhala-braun-sat-30-4-01.shuffled.cnf
C20	ezfact48_1.shuffled.cnf

Chapter 5

A Fast Approximation Algorithm for MIN-ONE SAT

In this chapter [33], we will propose a novel approximation algorithm for MIN-ONE SAT, which is the first in this area to the best of our knowledge. This approximation algorithm can help to achieve a drastic performance improvement, thus making MIN-ONE SAT more feasible and affordable.

Our approach is fundamentally different from the existing OPTSAT and PBSAT algorithms described in Chapter 2. Instead, it is an approximate algorithm, which means that the solution obtained may not be guaranteed to be optimal. It should be pointed out that our algorithm holds its value on fast-solving because complete algorithms have to implicitly search all solutions to find the optimal one. Such a complete search may become an excessive burden given the sheer size of solution set. Our approximation algorithm can potentially skip many search spaces, hence the computational complexity and resource usage can be significantly reduced. In addition, for many applications a low-cost near-optimal solution is more meaningful than the costly optimal solution, where our algorithm becomes a perfect fit. We call our approach as RelaxSAT because it uses a relaxation technique.

The detailed algorithm was exhibited in Section 5.1. We will also demonstrate the outcome of

this algorithm in Section 5.2.

5.1 Our Approach

5.1.1 RelaxSAT

The basic idea of the proposed RelaxSAT is to generate a set of constraints from the objective function to guide the SAT solver. The detailed algorithm is shown in Listing 5.1. A set of constraints, S_c , is first generated in order to minimize (or maximize) the objective function. In other words, the constraint-set defines a solution space within which the optimal value of the objective function is reached. The constraints are unit clauses that determines the corresponding variable assignments. For instance, in the basic MIN-ONE SAT, the initial constraint-set, S_C , is the set of unit negative-literal clauses that forces each variable to be ZERO. Note that it is trivial to construct S_c when the objective function is a linear function of the variables. This constraint set can be represented as a conjunction of all such negative-literal clauses, resulting in a CNF constraint formula F_{sc} . It should be pointed out that F_{sc} contains at most N clauses when every variable is constrained. Then, $F_A = F_{sc} \cdot F_o$, and F_A is passed to a SAT solver to solve. One may raise the concern that this kind of constraints set may cause considerable conflict(s) with the original Boolean formula F_o . While this may be true, we want to take advantage of these conflicts to benefit our algorithm. Whenever one or more conflicts exist between F_{sc} and F_o , a relaxation procedure is invoked. This relaxation procedure analyzes the reason behind the conflict(s) through the trace information provided by the backbone SAT solver and removes the constraints that are most responsible to the conflicts. Generally speaking, the constraints that cause conflicts will be removed gradually, thereby *relaxing* the constraints along the way.

Listing 5.1: RelaxSAT Algorithm

```
def RelaxSAT( $F_o$ /*input formula*/)
begin
     $i=0$  /*iteration counter*/
     $S_c = \text{InitConstrantSet}(\text{Obj}/*objective function*/);$ 
```

```

while ( )
     $F_A = F_o \cdot F_{sc}$ 
    Result=Call_SAT_Solver( $F_A$ )
    if (Result == SATISFIABLE)
        return /*solution found*/
    else
         $S_c = \text{RelaxConstraints}(S_c)$ ; /*Relaxation*/
end

```

Assume that F_o is satisfiable (otherwise, conventional SAT solvers are sufficient to determine its unsatisfiability), whenever F_A (i.e., $F_o \cdot F_{sc}$) is unsatisfiable, we can conclude that some clauses in F_{sc} are the reason that makes F_A unsatisfiable. In other words, whenever F_A is unsatisfiable, we can identify at least one unit clause in F_{sc} responsible for it. This allows us to formulate Theorem 4:

Theorem 4. Let F_o be satisfiable. If $F_A = F_{sc} \cdot F_o$ is unsatisfiable, the UNSAT core [80] contains at least one clause from F_{sc}

Proof. We prove this by contradiction. If the UNSAT core does not have any clause from F_{sc} , all the UNSAT core clauses will have to come from F_o . This leads to the conclusion that F_o is unsatisfiable, which conflicts with the assumption that F_o is satisfiable. \square

The relaxation procedure will be repeated until a solution is found. The number of iterations is bounded by the number of variables in F_o , as given in Theorem 5. Note that in Theorems 4 and 5, the same symbol annotations as Listing 5.1 are used.

Theorem 5. RelaxSAT will terminate within N iterations, where N is the number of variables in the formula

Proof. From Theorem 4, in each iteration at least one constraint clause in F_{sc} will be removed. Since the maximal size of F_{sc} is N , so in the worst case, after N iterations F_{sc} will become empty, indicating that at this point $F_A = F_o$. \square

Theorem 5 gives an upper bound on the computational complexity. Although up to N SAT iterations may be needed, the search space for many of these iterations is generally very small, due to the constraints added. Details of the computational complexity will be furthered in Section 5.1.3

To give the readers a more intuitive understanding of how RelaxSAT works, we will explain it in terms of search space enlargement. The initial solution that falls within constraint-set S_c actually represents a dot in the entire search space. Note that the constraints set can also define a subspace which contains multiple solutions. We call the corresponding representation in the search space as the *exploring space*. Thus, initially, the exploring space can be extremely small as potentially every variable is constrained. In each iteration, when one or more constraints are removed, the exploring space is enlarged. Whenever the exploring space does not intersect with solution space, F_A would be unsatisfiable. So the exploring space will keep growing until it eventually intersects with solution space, where a solution within the intersection of the exploring space and the solution space will be found. Figure 5.1 shows the exploring space enlargement process. The outer oval represents the entire space. The exploring space grows from a single dot in iteration I. The patterned areas represent the solution space. In iteration I, the exploring space does not overlap with solution space. So the algorithm continues to the next iteration where a overlapping space exists, shown as a red region in the figure.

One could observe that the exploring space behaves like those local search algorithms, where the search may be trapped in local optimal points and may not always find the global optimal. Thus, the final solution highly depends on the way the constraints are relaxed. To achieve a tight bound and efficiency the relaxation heuristic should be carefully crafted, which is discussed next.

5.1.2 Relaxation Heuristic

The basic principle of our relaxation heuristic is that the constraints that cause more conflicts should be relaxed first. The underlying assumption of this heuristic is that by removing those constraints that are more responsible for the conflict(s), we can reach a solution faster. Based on this principle, each constraint carries a *score* which represents how many conflicts it is involved.

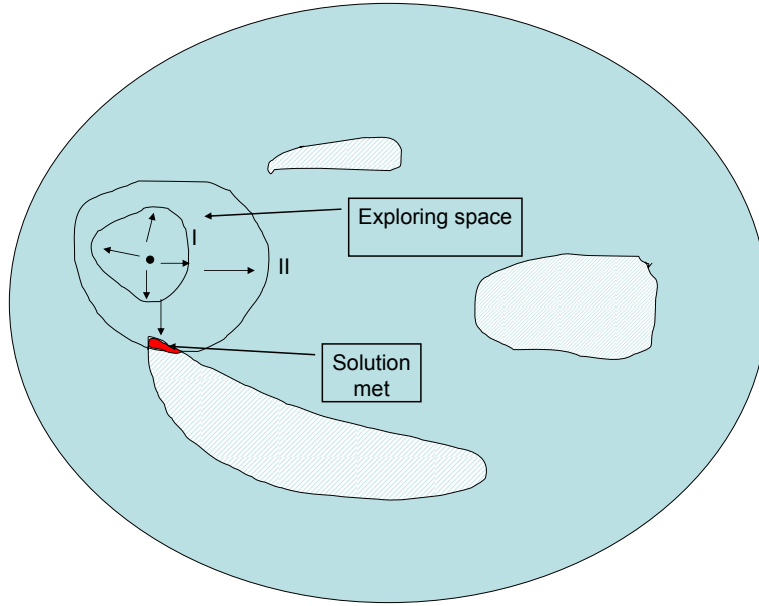


Figure 5.1: Exploring space enlargement

This conflict-score is computed in each iteration through the trace file for the UNSAT core provided by the SAT solver¹. From these trace files, the implication tree that proves the formula unsatisfiable can be recovered [78]. Given the recovered implication tree, a backward search can be performed from the conflict to identify the conflict-score of each constraint.

A simple example shown in Figure 5.2 explains how the proposed heuristic works. In Figure 5.2, two conflicts C1 and C2 rise due to the unit clause constraints $(\bar{x})(\bar{y})(\bar{z})$. Based on a backward traversal, we identify that constraint \bar{y} is involved in both conflicts, C1 and C2, while the other two only contribute to one conflict. Thus the conflict-scores of $(\bar{x})(\bar{y})(\bar{z})$ are $\{1, 2, 1\}$. Using the obtained conflict-scores, constraint (\bar{y}) will be chosen to be relaxed in this iteration.

It is not necessary to relax only one constraint at each iteration. To reduce the number of iterations, in our implementation we relax the top ten-percent of constraints on the conflict-score list. For those variables with tied conflict-scores, we order them with a favor on those appear less

¹many modern SAT solvers are capable of providing the trace file for the unsatisfiable instances, such as ZChaff, Minisat[30]

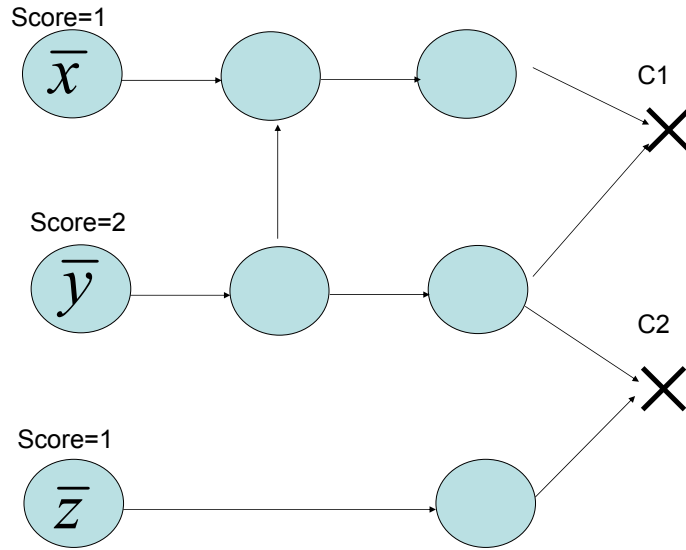


Figure 5.2: Conflict guided relaxation heuristic

frequently in the formula. It should be noted that if more constraints are relaxed in each iteration, the quality of the final solution may be degraded due to the decreased granularity in the search. For the MIN-ONE SAT extension, where each variable is associated with a coefficient, we pick the constraint that has the smallest coefficient and the highest conflict-score in order to maintain minimal loss when trying to satisfy the objective function.

A side benefit should be mentioned here. The returned solution from RelaxSAT on MIN-ONE SAT problems could have fewer ONES than the constraint-set specified. For example, given a formula with 200 variables, if a solution is provided by RelaxSAT under a constraint-set with 100 variables specified as ZEROS, the upper bound of ONES would be $200 - 100 = 100$. However, during the search, the SAT solver may assign the (unconstrained) free-variables to ZERO, leading to an even smaller number of ONES.

The experiments in Section 5.2 demonstrates that our heuristic works very well to obtain a tight bound while keeping a low computational overhead.

5.1.3 Discussion on Complexity of RelaxSAT

In this subsection, we will discuss briefly about the algorithm complexity. From Theorem 5, we know that in the worst case RelaxSAT may require N iterations. In each iteration the original Boolean formula F_o conjuncted with some unit-clause constraints F_{sc} will be solved. Although theoretically the SAT algorithm is NP-complete, modern SAT solvers can solve many of the large instances without explosion in time or space. Based on these facts, conventional SAT solvers have much less computational cost when compared with existing MIN-ONE SAT solvers. In this regard, because RelaxSAT relies on conventional SAT, in each iteration its complexity stays at the baseline of SAT performance. Considering the maximum number of iterations is limited by N , practically the total computation complexity of RelaxSAT is much smaller compared with other existing MIN-ONE SAT algorithms. The memory overhead of RelaxSAT is also in line with conventional SAT because only an extra constraint-set is introduced, which is simply a set of at most N unit clauses. The unit clauses actually help to reduce the search space of F_o by forcing some variables to a fixed value, especially in the earlier iterations. Generally RelaxSAT consumes significantly less resources compared with PBSAT based algorithms where an additional Boolean network F_{sc} is needed, without any fixation of any variables.

5.2 Experimental Results

Our proposed RelaxSAT was implemented in C++ under 32-bit Linux, and all the experiments were conducted on a Intel Xeon 3.0G workstation with 2G memory. ZChaff version 2004.11.15-simplified was chosen to serve as the backbone SAT solver in RelaxSAT due to the convenience on obtaining the trace information. There are three categories of benchmarks in our experiments. The first category is from the OPTSAT benchmarks, where OPTSAT outperformed some popular PBSAT based solvers [38]. Since OPTSAT outperformed PBSAT for these benchmarks, in the first experiment we only compared RelaxSAT with OPTSAT. Note that the publicly distributed OPTSAT binary program can only solve the basic MIN-ONE SAT problem, so it is not compared in the second and third experiments, where the extended MIN-ONE SAT problem was targeted.

We report the results of the first experiment in Table 5.1. For each benchmark listed under the first column, the number of variables and clauses of each instance are listed in column two. The third and fourth columns present the run time of OPTSAT and RelaxSAT, followed by a column of run time speedup. The speedup is defined as the ratio between the run time of OPTSAT and RelaxSAT. The objective function bound returned from OPTSAT and RelaxSAT are reported in the sixth and seventh columns. The last column shows the difference between the number of ONES OPTSAT and RelaxSAT obtained. Note that whenever OPTSAT is able to solve the instance, it is guaranteed to be the optimal solution. In Table 5.1, OUT indicates OPTSAT times out after 3600 seconds and the corresponding run time speedup is recorded as INF. Whenever OPTSAT times out, the sixth and last columns are marked X.

In 12 of the 21 benchmarks, both OPTSAT and RelaxSAT found the optimal solutions. But RelaxSAT achieved a two to three times speedup. For 4 of the remaining instances, OPTSAT failed to finish in 3600 seconds while RelaxSAT can provide a solution in only a few seconds. Relax SAT returned tight bounds on the remaining 5 benchmarks. For very few instances, the run time is slightly increased due to the large number of iterations that RelaxSAT went through.

The second experiment involves the benchmarks from the power-aware test pattern generation, where the goal is to generate a test pattern can detect a target delay fault and simultaneously excite as much switching activities as possible to worsen the delay effect. Since considering a complicated fault model is irrelevant with this chapter, in our experiments we only check the maximal switching activities of a sequential circuit in two consecutive clock cycles without any other constraints. In the experiment setup, we unroll each sequential circuit into a two-time-frame combinational circuit. The switching activities are monitored by a set of XNOR gates with the inputs from a corresponding gate pairs in two time frames. If the output of a monitor XNOR is ONE, it means there is no switching activity on that gate pair. After transforming the two-time-frame combinational circuit and the monitor circuit into a Boolean formula, obviously the task of maximizing the switching activities now is equivalent to finding a MIN-ONE solution on the outputs of the XNOR gates. It should be emphasized that this problem is different from the basic MIN-ONE SAT because in basic MIN-ONE SAT *all* the variables have to be considered while in

the switching activity maximization only the outputs of the monitor circuits are considered. In other words, the objective function only involves a portion of the variables in the Boolean formula.

In this second experiment, RelaxSAT is compared with MiniSAT+ [31]. MiniSAT+ is a high-performance PBSAT solver which can be used to solve MIN-ONE SAT. The MiniSAT+ is chosen not only because it has been demonstrated to have high performance [3], but also due to reason that its techniques favor those PBSAT problems where most of the constraints can be directly represented as single clauses. The usual MIN-ONE SAT instances fall into this category and can be benefited by using MiniSAT+, because all the constraints in MIN-ONE SAT are SAT clauses except the objective function.

The results of this second experiment are shown in Table 5.2. The eight columns of Table 5.2 are similar to Table 5.1, except that OPTSAT is replaced by MiniSAT+ due to reason that OPTSAT is unable to handle the extended MIN-ONE SAT problems. Since MiniSAT+ works iteratively, whenever it cannot find the optimal solution within 3600 seconds, it reports the best solution obtained so far in the third column. A positive value in the last column indicates RelaxSAT returned a **tighter** bound. Among the 9 instances, RelaxSAT outperformed MiniSAT+ in 5 of them with better result and shorter run time. One can see that the run time of RelaxSAT is roughly **40** \times smaller than that needed by MiniSAT+. For example, in s5378, RelaxSAT can return a solution containing 1075 non-switching gate in less than 21 seconds while MiniSAT+ generates 167 more non-switching gates even after one hour. For some small instances, MiniSAT+ was able to find the global optimum, the solution obtained by RelaxSAT still presents a very tight margin.

In the third experiment, we evaluated RelaxSAT on a set of benchmarks whose objective functions have integer coefficients associated with each variable. Some of them were generated similarly as those in the second experiment. The only difference is that here the gate size is considered when computing the gate switching activities. Thus the objective functions will be slightly different, where the variables representing the outputs of the XNOR gates will have a positive coefficient corresponding to the gate size. Actually this is a more accurate model in the power consumption estimation. The gate size information is obtained through the synthesized circuits. The rest of the

benchmarks in this experiment are derived from the OPTSAT benchmarks with a randomly generated coefficient of each variable. So the objective function becomes *minimize* $\sum_{i=1}^N c_i \times V_i$. Note that c_i is a number between -20 and 20.

The layout of Table 5.3 is the same as Table 5.2. RelaxSAT outperformed MiniSAT+ on 6 of the total 12 benchmarks. The remaining 6 instances are small ones except for s6669 and b15. For s6669 and b15, MiniSAT+ was able to find a better solution than RelaxSAT, although it could not produce the global optimum within 3600 seconds. The run times shown in the third column of these two circuits are the time needed to get a better solution than RelaxSAT. Here we take a closer look at b15 to discuss the strength and potential improvement of RelaxSAT. For b15, RelaxSAT was able to finish after 274 seconds with the result of 27748. Meanwhile MiniSAT+ discovered a better solution after more than 1800 seconds but could not step further until it finally timed out at 3600 seconds. One can see that although in this specific example RelaxSAT was not able to outperform MiniSAT+ in terms of the tightness of the bound, RelaxSAT holds the strength on a much smaller run time. In other words, MiniSAT+ needed three times more execution time to outperform RelaxSAT by a small margin. This observation suggests that using RelaxSAT as a preprocessing step before MiniSAT+ may be beneficial, which could be a potential future extension of RelaxSAT.

The last experiment was set up to demonstrate the low memory overhead of RelaxSAT, compared with both OPTSAT and MiniSAT+. A total of ten large instances are selected to measure the peak memory consumption during the solving process. The results are presented in Figures 5.3 and 5.4. In both figures the X-axis denotes the instance and Y-axis lists the memory consumption in megabytes (MB). In Figure 5.3, five instances are selected from Experiment I and the memory consumption are compared between OPTSAT and RelaxSAT. It is clear that RelaxSAT has a smaller memory cost especially on large instances like bmc-galileo-8. The memory cost of five instances from Experiment III are shown in Figure 5.4. Similar to Figure 5.3, RelaxSAT required a significantly smaller memory consumption compared with MiniSAT+. The reason is as we mentioned before: RelaxSAT only has an additional constraint-set while PBSAT based algorithms need a large Boolean network of F_c .

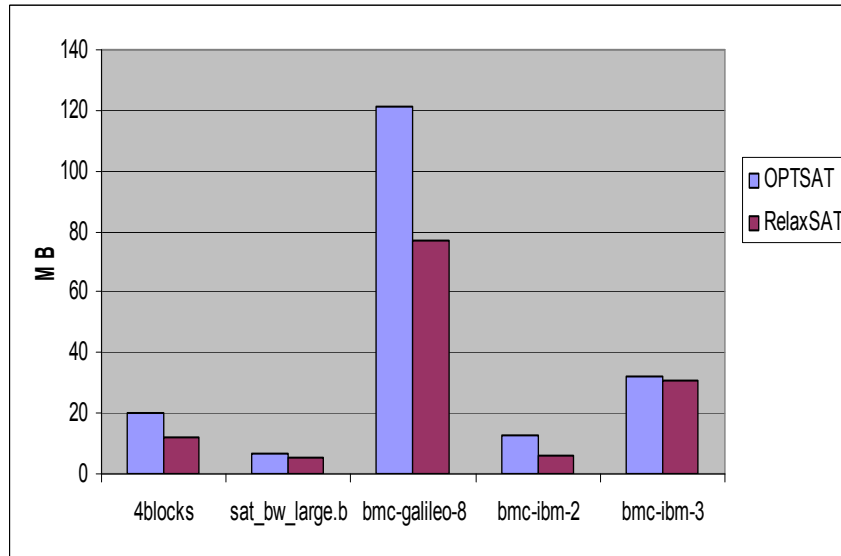


Figure 5.3: Memory consumption: OPTSAT vs. RelaxSAT

5.3 Summary

In this chapter, a novel approximation algorithm for MIN-ONE SAT is proposed. A set of constraints automatically derived from the objective function is used to guide the search. Whenever the constraint-set causes conflicts with the original Boolean formula, those constraints most responsible to the conflicts will be identified and relaxed. The relaxation procedure continues until a solution is found. Our proposed algorithm has a low memory overhead and can provide a tight bound for the objective function. It is able to handle some large instances that the existing MIN-ONE SAT solvers failed. Meanwhile it can achieve one or two orders of magnitude run time reduction.

In the future, different relaxation heuristics can be explored, and a possible hybrid solution can also be investigated. Although RelaxSAT can provide a tight bound with a significant performance improvement, it may not guarantee a global optimum. To address this problem, we believe a hybrid solution of RelaxSAT and other complete MIN-ONE SAT algorithm can be very beneficial. One possible approach is to use the RelaxSAT as a preprocessing step in PBSAT based algorithms

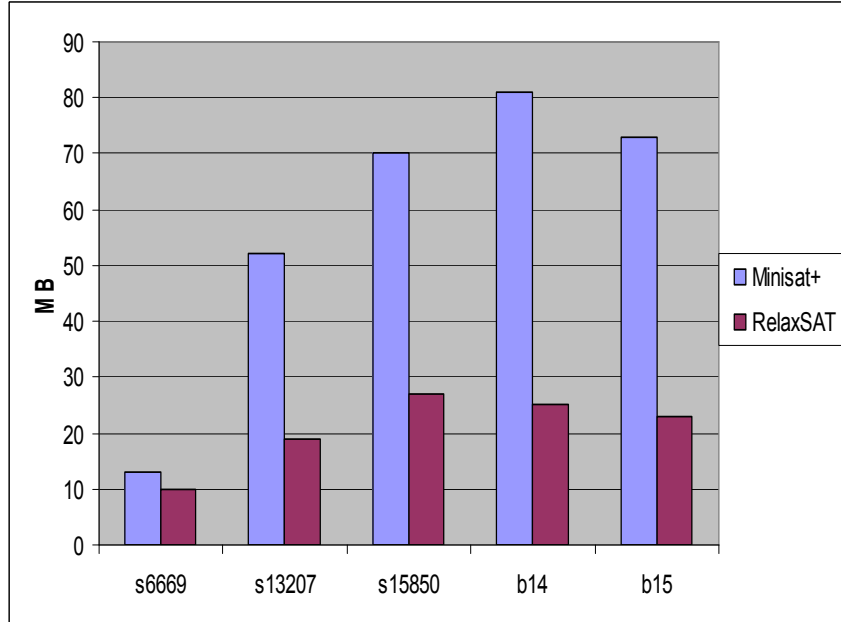


Figure 5.4: Memory consumption: MiniSAT+ vs. RelaxSAT

where the bound estimation BE is passed from RelaxSAT. In this hybrid scheme, RelaxSAT would be able to provide a tight initial bound estimate quickly, allowing PBSAT-based algorithms to skip many unnecessary iterations.

Table 5.1: Experiment I: OPTSAT Benchmarks

CNF	#Var/#Cls	OPTSAT TIME	RelaxSAT TIME	SpeedUp	OPTSAT result	RelaxSAT result	DIFF
c17.cnf	13/22	0.02	0.01	2.00	4	4	0
c880.cnf	469/1164	OUT	0.02	INF	X	198	X
2bitcomp_5	125/310	2.16	0.01	216.00	39	45	-6
2bitmax_6	252/766	333.88	0.11	3035.27	61	68	-7
3blocks	283/969	0.62	0.31	2.00	56	61	-5
4blocks	758/47820	OUT	6.34	INF	X	116	X
4blocksb	410/24758	1.06	0.68	1.56	66	66	0
qg1-08	512/148957	52.66	37.78	1.39	64	64	0
qg2-08	512/148957	31.4	16.64	1.89	64	64	0
qg3-08	512/10469	0.31	0.47	0.66	64	64	0
s27	72/141	0.02	0.01	2.00	24	24	0
s5378	12168/26527	OUT	97.89	INF	X	2796	X
sat_bw_large.a	459/4675	0.02	0.28	0.07	73	73	0
sat_bw_large.b	1087/13772	0.22	1.5	0.15	131	131	0
sat_logistics.a	728/5784	1.51	0.68	2.22	135	135	0
sat_logistics.b	757/6429	4.38	0.76	5.76	138	138	0
sat_rocket_ext.a	331/2246	1.4	0.16	8.75	65	65	0
sat_rocket_ext.b	351/2398	1.68	0.18	9.33	69	69	0
bmc-galileo-8	58074/294821	OUT	497.31	INF	X	12303	X
bmc-ibm-2	2810/11683	1013.13	10.64	95.22	940	1017	-77
bmc-ibm-3	14930/72106	22.37	92.83	0.24	6356	6362	-6

OUT=3600 sec

Table 5.2: Experiment II: Power-Aware Test Generation Benchmarks

CNF	#Var/#Cls	MiniSAT+ TIME	RelaxSAT TIME	SpeedUp	MiniSAT+ result	RelaxSAT result	DIFF
s510	972/2254	0.24	0.23	1.04	122	138	-16
s1494	2720/6812	9.25	2.15	4.30	342	421	-79
s5378	12168/26508	3600	20.85	172.66	1242	1075	167
s6669	13828/31152	3600	53.45	67.35	1635	1553	82
s15850	41880/89908	3600	402.51	8.94	6265	6259	6
s35932	72592/164716	3600	1131.21	3.18	8198	7456	742
b13	1448/3248	26.6	0.43	61.86	157	211	-54
b14	40392/98250	3600	458.99	7.84	5857	5785	72
b15	35688/87808	3600	281.97	12.77	6423	6624	-201

Table 5.3: Experiment III: Objective Functions with Integer Coefficients

CNF	#Var/#Cls	MiniSAT+ TIME	RelaxSAT TIME	SpeedUp	MiniSAT+ result	RelaxSAT result	DIFF
2bitcomp_5	125/310	0.94	0.01	94.00	-297	-203	-94
3blocks	283/969	3.39	0.34	9.97	-105	-81	-24
qg3-08	512/10469	1	0.39	2.56	-73	-29	-44
sat_bw_large.a	459/4675	0.34	0.2	1.70	-189	-189	0
sat_logistics.b	757/6429	3600	0.86	4186.05	-197	-269	72
bmc-ibm-2	2810/11683	3600	9.86	365.11	-616	-581	-35
s5378	12168/26508	3600	18.9	190.48	4570	3888	682
s6669	13828/31152	86.27*	180.65	0.48	6538	6732	-194
s13207	35088/74658	3600	243.07	14.81	14942	14248	694
s15850	41880/89908	3600	359.48	10.01	19394	19244	150
b14	40392/98250	3600	727.27	4.95	23174	21144	2030
b15	35688/87808	1821.3#	273.59	6.66	27392	27748	-356

*:after 3600 seconds, the best result is 6386

#: after 3600 seconds, the best result is still 27392

Chapter 6

A RelaxSAT based MAX-SAT Solver

Due to its theoretical and practical importance, the MAX-SAT solver is receiving increased levels of attention. After our previous work on the MIN-ONE SAT solver, a new direction for MAX-SAT could be explored along a similar manner. To validate this, we built a MAX-SAT solver on top of our novel MIN-ONE SAT algorithm (RelaxSAT). By taking advantage from the relaxation heuristic inside RelaxSAT, those clauses that potentially contribute most to the conflicts will be identified in order to ease the efforts to satisfy the rest of the formula. This local greedy scheme helps us to cover the possibly largest satisfiable part of the formula, which is exactly what the MAX-SAT solver attempts to do. In this chapter, we introduce a new MAX-SAT solver (RMAXSAT) based on our previously proposed MIN-ONE SAT algorithm. From the experimental results our solver achieves a significant performance improvement over existing MAX-SAT solvers.

Similar to the conventional SAT solver, the MAX-SAT solver also has a great potential to find its applications in formal verification. In Section 6.3 We also discussed how to apply our MAX-SAT solver on bounded model checking. To the best of our knowledge it's the first scheme that has ever been proposed to deploy MAX-SAT in bounded model checking. Three possible applications of RMAXSAT are described and some corresponding algorithms are outlined.

6.1 The New MAX-SAT Solver

Because our MAX-SAT Solver is based on the RelaxSAT proposed in Chapter 5, we name it RMAXSAT. RMAXSAT consists of three components. The first component is formula transformation. A RelaxSAT implementation is also embedded in RMAXSAT and finally the MAX-SAT solution will be taken from the outputs of RelaxSAT and printed out. The pseudo code is presented in Listing 6.1. The program of RMAXSAT was implemented in C++ under Linux.

Listing 6.1: Our Solver: RMAXSAT

```
def RMAXSAT(F)
begin
    F_n=transform(F)
    Solution=RelaxSAT(F_n)
    return solution //contains both assignment and bound
end
```

We would mention a few details about the implementation of RMAXSAT here. As previously discussed in Section 5.1.2, in the underlying MIN-ONE SAT algorithm, it is not necessary to relax only one constraint at a time, more constraints can be relaxed to reach a solution faster. However, in RMAXSAT, to achieve the bound as tight as possible, we relax only one constraint in each iteration. Because we want a MIN-ONE solution on the auxiliary variables only, the MIN-ONE SAT target function passed to RelaxSAT algorithm is $\sum_i^M AX_i$, where AX_i is the i^{th} auxiliary variable that is embedded in the i^{th} clause with a total of M clauses in the original formula. RMAXSAT can be easily extended to handle partial MAX-SAT problems, where the auxiliary variables are not inserted to every clause in the formula. By adding auxiliary variables only to a subset of clauses, those clauses without auxiliary variables will automatically be forced to be satisfied during the search for a solution.

Since the RMAXSAT inherits the search heuristics from RelaxSAT, it intends to leave those clauses that would most likely cause the conflicts alone (by setting their corresponding auxiliary variable to one) and satisfy the rest of the clauses in an iterative manner. Our experiments have

shown this conflict-guided relaxation heuristic works effectively in most cases.

First let us look at a case study of RMAXSAT. We have tested RMAXSAT on some equivalence checking problems to see if RMAXSAT can correctly identify the one clause, that when removed, can make the rest of the clauses satisfiable. As discussed at the beginning of this dissertation, equivalence checking compares two different designs to check whether they are functionally equivalent or not. By adding the extra equivalence assertions, the satisfiability of the equivalence-checking instances determines the equivalence between the two designs. When the two original designs are indeed equivalent, the equivalence-checking instance should be unsatisfiable. In other words, an effective MAX-SAT solver should be able to identify the equivalence assertion clause, without which the rest of the clauses should be satisfiable.

Thus, in our preliminary setup, when an unsatisfiable instance derived from an equivalence checking problem is fed to RMAXSAT, it is quite a surprise to us that RMAXSAT was able to quickly find a MAX-SAT solution that satisfies the clauses from the design without the clause from the equivalence assertion. This is extremely encouraging because logically the best way to make an unsatisfiable equivalence-checking instance to become satisfiable is by removing the equivalence assertion constraints. This observation gave us the hints that RMAXSAT might be a powerful tool to reveal some insightful reasonings encoded in the SAT instances. This leads us to explore the potential to apply RMAXSAT on some hard formal verification problems like bounded model checking, which is discussed in Section 6.3.

6.2 Evaluation of MAX-SAT Solver

Before the discussion about the applications of RMAXSAT, in this section some experiments were conducted to compare RMAXSAT with some existing MAX-SAT solvers. All the experiments were conducted on a PC workstation with Intel Xeon 3.2G CPU and 2G bytes of memory.

First we cite the experimental results from [38], presented in Table 6.1. The reason we cite this table here is that we want to demonstrate that OPTSAT and MSAT+ are the state-of-the-art

solvers which have a significant edge among the existing MAX-SAT solvers. Since the source code or executable files from most MAX-SAT solvers are not available to us, at current stage we first compare the performance between our RMAXSAT and OPTSAT, which is shown in Table 6.2. The comparison with MSAT+ is reported in Table 6.3. By evaluating on the same set of publicly available benchmarks, we can check to see if RMAXSAT is indeed a high-performance MAX-SAT solver.

In Table 6.1, the first column lists the names of the benchmark formula in the format of CNF. The maximal number of satisfied clauses are shown in the second column for each instance, which is the optimal bound if at least one of the deterministic MAX-SAT solver should return if it is able to finish. The running time of each MAX-SAT solver in the competition is presented in the following columns, in the order of BF [13], OPBDP [18], PBS4 [5], MSAT+ [3][31]. Solver BF is a MAX-SAT Solver while OPBDP, PBS4 and MSAT+ are generic pseudo Boolean solvers. MSAT+ is a pseudo Boolean SAT solver based on MiniSat [30] and it is shown to be capable of solving large number of instances.

All of the deterministic MAX-SAT solvers can report the optimal bound if they could finish within the time and memory limits. The timeout limit is set at 1800 seconds while the memory consumption is limited within 1 GB. “TIME” in the table indicates that the corresponding solver times out. Similarly “MEM” indicates memory is exhausted. Note that “SF” means the solver aborted unexpectedly. When no solver can solve the instance, the optimal bound in the second column is marked as “N/A”. Finally “-” represents the solver returns wrong results. It can be observed that solver BF, OPBDP and PBS4 were not able to solve most of the instances, either due to the resource limit or the implementation bugs. On the other hand, MSAT+ and OPTSAT solved all the instances except u-bw.b. For instance, in the instance c3540, BF timed out, OPBDP ran out of memory, PBS4 returned wrong results, both MSAT+ and OPTSAT finished in 242.02 and 786.33 seconds, respectively. The maximum number of clauses that could be satisfied is 9325. From this table we see that the two solvers, MSAT+ and OPTSAT, gave the best performance and capability edge over other three MAX-SAT solvers. Therefore, in the following experiments, we compare our proposed RMAXSAT with OPTSAT and MSAT+ separately.

Table 6.2 and Table 6.3 share the same layout, where the first column exhibits the instance name, followed by the results from two solvers in the comparison. In Table 6.2, the second and the third columns report the bound and running time of OPTSAT, while the fourth and fifth columns show the bound and running time of RMAXSAT. The sixth column presents the bound difference between RMAXSAT and OPTSAT, computed as the bound obtained from RMAXSAT minus the bound obtained from OPTSAT. It should be noted that since RMAXSAT is based on the approximation algorithm of MIN-ONE SAT, the bound it returned may not be optimal. Finally, in the last column, the speed up is calculated as $\frac{RunTimeofRMAXSAT}{RunTimeofOPTSAT}$. We set the time out limit the same as in Table 6.1, 1800 seconds.

One can see that the bounds returned by RMAXSAT are extremely tight. In all 27 instances, RMAXSAT reported optimal bounds on 18 of them. In the remaining 9 instances, the difference is at most 5, usually it is only 1 or 2. Meanwhile the performance improvement is over 70 times on average. For example, in qvar14, OPTSAT obtained the MAX-SAT result of simultaneously satisfying 9312 clauses in 188.09 seconds, while RMAXSAT obtained the same MAX-SAT bound of 9312 in only 1.06 seconds. This is a speedup of $177.44\times$. In another example, RMAXSAT finished instance u-bw.b in just 1.14 seconds whereas all the other MAX-SAT solvers failed. With this result we know that in u-bw.b.cnf, at least 11480 clauses can be satisfied simultaneously, which means by removing only one clause the formula becomes satisfiable.

The results from Table 6.3 follow the same trend as in Table 6.2, and on average, over 50 times performance improvement from RMAXSAT was achieved over MSAT+. For example, in the instance u-log.c, MSAT+ finished the instance in 94.13 seconds that simultaneously satisfied 9506 clauses. RMAXSAT, on the other hand, satisfied the same number of clauses in just 0.39 seconds. This is a $241\times$ speedup. Based on the consistent results obtained in both experiments, it is safe to conclude that RMAXSAT is able to return tight bounds with performance boost with over an order of magnitude reduction in the computational cost.

6.3 Applications of RMAXSAT in Bounded Model Checking

We have demonstrated the power of our proposed RMAXSAT. In this section, we will discuss the potential applications that can benefit from RMAXSAT.

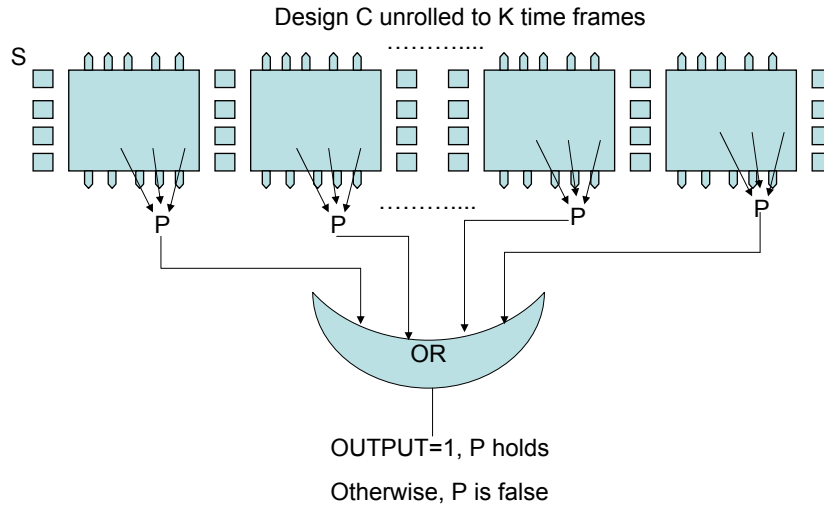


Figure 6.1: Basic Bounded Model Checking Setup

First, let us quickly review the set up of Bounded Model Checking (BMC). In BMC, the sequential circuit C is unrolled to the target depth K , resulting in the unrolled circuit CU . A property P will be verified on the unrolled circuit CU . Considering the property P to be an invariant, P should be checked whether it holds in each time frame. Given the initial states S of the circuit, BMC verifies whether there exists a path of length K from the initial states S in the state transition graph that can violate the property along any path of length K . Note that S could be a single fully-specified state or a set of states. Such a basic setup of BMC of invariant properties is shown in Fig 6.1. The big OR gate is inserted to test whether P is falsified at any time-frame. If the BMC is conducted in an incremental way, the basic BMC setup can be enhanced as illustrated in Fig 6.2,

where P is held through the path except the last time frame. In the last time frame, $\neg P$ is asserted to see whether a counter example can be found. The detailed explanations of these BMC setups can be referred at [26][27],[63].

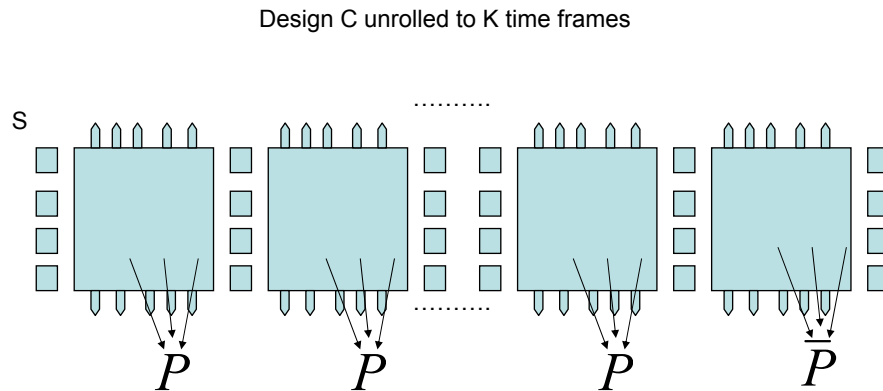


Figure 6.2: Enhanced Bounded Model Checking Setup

The unrolled BMC setup can be converted into a representation of either Binary Decision Diagram (BDD) or a formula of satisfiability [26]. Both the BDD and SAT formulations can be solved with various techniques. Since the construction of BDD causes memory blowout easily, the SAT formulation for BMC is extremely popular. Note that the SAT translation of a circuit is linear with the size of the circuit in terms of computational complexity, which is actually trivial compared with the time for solving the instance. After the whole BMC setup is translated into a formula F , the SAT solver is invoked on F . If F is satisfiable, then a counter example has been found that violates P and the solution of F presents the path that leads to \bar{P} from S . On the contrary, the unsatisfiability of F means that P holds at least at the current unrolling length from state S .

In the incremental BMC setup, the property P will be verified gradually from 1 time-frame to a

preset K time-frames. Whenever a counter example is found, the search stops and P is found to be falsifiable. Otherwise, BMC will continue until K time frames are reached or the sequential depth of the circuit is met [68][61]. Let us imagine that we apply RMAXSAT on the scenario where the P holds under the unrolling length. Since P holds within the depth K , formula F would be unsatisfiable. By utilizing the intelligent search heuristics from RMAXSAT, we can find a solution Q that satisfies most of the clauses in F . Now we are able to identify those left-over clauses and correspondingly these clauses can be interpreted as the *reason* that causes the unsatisfiability. One may raise the concern that the reasoning from RMAXSAT might not make too much sense in BMC since the circuit structure could be destroyed when some of the clauses representing the circuit structure are removed. We understand this concern and will discuss how this very important issue is addressed.

Because the circuit structure should be retained completely, the only reason that can potentially cause the unsatisfiability is the initial states S . In other words, S and the Boolean constraints from P CANNOT be satisfied simultaneously under the hard constraints (clauses) from the circuit structure. Obviously when both the circuit structure and P are kept, we have to remove some constraints from S to conceivably satisfy F . This is where RMAXSAT can be deployed. It should be pointed out that RMAXSAT has to be modified to be able to understand the difference between clauses from the circuit and clauses from the initial state(s), where only the clauses from the initial state(s) are allowed to be unsatisfied. In fact, this is a Partial MAX-SAT problem that the circuit clauses are hard constraints and the initial-state constraints are soft. Since it is trivial to modify RMAXSAT to solve Partial MAX-SAT, we do not discuss the details here.

Now with the RMAXSAT that supports the Partial MAX-SAT solving, we are able to find a solution of F that satisfies the whole formula except for some clauses from the initial state. Usually the initial states S are constituted by a set of unit clauses. Then this can be understood as by removing some initial-state constraints property P can be violated (the formula becomes SAT) within K depth. One can interpret it as the state strengthening on the initial states because the initial states are less constrained under the solution returned from RMAXSAT. We view this as a powerful reasoning technique that has never been proposed in the past. The detailed analysis will

be discussed in the following.

When RMAXSAT returns a solution of F , some of the constraints on the initial state would have been removed. In other words, in order to find the MAX-SAT for F , some of the constraints on S would have to be relaxed. The assignments on both those relaxed and unrelaxed initial-state variables define a new set of initial states. We name these initial states S' and it is easy to see that $S \subseteq S'$. Since S' contains some states that can violate the property within depth K , these states are deemed valuable when searching for a counter example. If any state in S' can be reached from S , then it is guaranteed that a counter example exists to prove that P is false. More importantly, states in S' might provide a low-cost search to reach the property, due to the underlying reason from RMAXSAT that the clauses which strongly attempt to induce conflicts will be abandoned first. Other state strengthening techniques usually involve image computation [70][40][42]. In these techniques, the preimage of the property is built to enlarge the target states, which is very expensive in terms of computation and storage complexities. On the other hand, with the help of RMAXSAT, it is able to obtain a subset of states than can reach the property in a fractional cost of the preimage computation. Note that S' may not simply be a subset of the superset of the preimage, where they could intersect with each other. Due to the aforementioned reason, the computation of S' from a MAX-SAT solver could significant benefit the search for a counter example to a target property.

We elaborate a few possible applications of about the applications of S' in bounded model checking:

1. S' can be used to guide the simulation-based property checking where the random inputs are generated by the information on the state elements. Since we know which state elements are relaxed by RMAXSAT, these states elements should be favored in the input generation.
2. In some abstraction-refinement schemes, we need to know which state element should be abstracted. Similar to the first item, S' can bring some insightful hints on this.
3. Since S' is a subset of preimage of the property, it could be incorporated into those preimage

based techniques, even within unbounded model checking.

When we look at the unrelaxed and relaxed state variables together, we will find that the relaxed state variables might only be assigned to the opposite value to the original values in S in order to violate P . To make it simple, let us consider an example $S = \{x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 0, x_5 = U, x_6 = U\}$ where x_1 to x_6 are state variables. “U” here means there is no constraints on that state variable. Hypothetically assume that after RMAXSAT, $S' = \{x_1 = 1, x_2 = U, x_3 = 0, x_4 = 0, x_5 = U, x_6 = U\}$. We can see that x_2 was relaxed to violate P . In fact, all those states to violate P can be only inside the sub-space where $x_2 = 1$. The reason is that a state inside S' with $x_2 = 0$ is S itself and S is proved to contain no state that can violate P . Similarly when multiple variables are relaxed, the last variable has been relaxed will hold the aforementioned property. More specifically, if we name the last relaxed state variable as V_r and its corresponding values in state set S or S' as $Value(V_r, S)$ or $Value(V_r, S')$, it is true that all states in S' that can falsify P have the property that $Value(V_r, S) = \neg Value(V_r, S')$. This claim is supported by Theorem 6.

Theorem 6. Given the aforementioned definitions of S , S' and V_r , for all the states that can violate P , the values of V_r on S and S' are opposite values: $Value(V_r, S) = \neg Value(V_r, S')$.

Proof. We prove this by contradiction. Suppose there is a state T can violate P with a single state variable V_r relaxed in the last iteration where $V_r(A, S) = V_r(A, S')$. Since we know that only one constraint was relaxed in the last iteration, in the iteration immediately before V_r was relaxed, condition $V_r(A, S) = V_r(A, S')$ holds. We assume this iteration as the i^{th} iteration. Due to the existence of T there should exist at least one solution to satisfy the formula in iteration i . But in iteration i we already know that the instance was unsatisfiable, otherwise RMAXSAT will have finished and V_r would not be relaxed. Thus, a contradiction. \square

There is another way to interpret the relaxation procedure in RMAXSAT. From Theorem 6 the relaxation on the state variables can be viewed as flipping on the values. Actually the state set S was altered by a sequence of flipping on the state variables until P is violated. In this regard, it is similar to the local search algorithms where neighbors are searched greedily to find the solution.

Here in each iteration the flipping on one state variable just moves S to one of its neighbors in the state space. Meanwhile the current sub-space was excluded to have any possibilities to violate P .

6.3.1 Discussions on RMAXSAT_ALL

Listing 6.2: Our Solver: RMAXSAT

```

def RMAXSAT_ALL( $F, P, S$ )
begin
    // $F$  is the CNF instance of BMC,
    // $S$  is the set of clauses representing the initial states,
    // $P$  is the set of clauses from the property.
     $S'_{ALL}=0$ 
    while ( $S$  contains at least one state element)
         $SLN=RMAXSAT(F,P,S)$  //get MAX-SAT solution
         $S'_{ALL}=S'_{ALL}\cap SLN$ .
         $F=block(F,SLN)$  //block solution  $SLN$  in  $F$ 
end

```

One may raise the concern that RMAXSAT only returns one solution thus S' is quite limited. This issue can be address by the algorithms presented in Listing 6.2. The basic idea is to force RMAXSAT working like a multiple-solution solver. Whenever a solution SLN is returned by RMAXSAT, it will be blocked in future iterations by adding a blocking clause for SLN to the clause database. Implicitly these blocking clauses will keep RMAXSAT searching until all the constraints from S are relaxed. This algorithm is called RMAXSAT_ALL and the solution set it returns is named S'_{ALL} . Note that the solution from RMAXSAT_ALL is different from the preimage of the property. The preimage of property P contains ALL the states that can reach P at depth K . On the other hand, S'_{ALL} only contains those states that are the “neighbors” of S which might reach P . Since at each iteration RMAXSAT_ALL relaxes as few state elements as possible, the solutions it obtains are generally have few bits of difference from S . It should be noted that the “neighbor” here not only indicates the Hamming distance on state elements but much more importantly it

reveals they are logically connected through our conflict relaxation reasoning. Consequently S'_{ALL} is relatively small in size.

Another interesting phenomenon we want to mention is that the solution SLN tends to be stretched to cover more state space after each iteration since more and more state elements will be relaxed until all of them are gone. The order and sequence of SLN in S'_{ALL} might help us to understand the bottleneck of proving P by analyzing the dependency between state elements.

We believe that S'_{ALL} has much potential to benefit bounded model checking, unbounded model checking and even reachability analysis, especially if the preimage computation is too costly and approximations are needed. Due to the reason that the research on this topic are still very preliminary, we will not address this in further detail in this dissertation.

6.4 Summary

We introduced a new MAX-SAT solver RMAXSAT in this chapter. The RMAXSAT was compared with other existing MAX-SAT solvers to show its performance strength. The potential applications of RMAXSAT in bounded model checking was discussed. A theorem was presented to reveal some interesting discoveries about the RMAXSAT for bounded model checking. Finally, an algorithm that attempts to iterate on RMAXSAT, RMAXSAT_ALL, was described and discussed that could benefit formal verification instances.

Table 6.1: Existing MAX-SAT Comparison

CNF	#C	BF	OPBDP	PBS4	MSAT+	OPTSAT
barrel3	941	0.23	2.04	0.88	0.12	0.9
barrel4	2034	0.65	47.59	11.67	0.34	21.19
barrel5	5382	21.42	MEM	MEM	24.01	177.11
barrel6	8930	213.6	MEM	-	95.56	896.45
barrel7	13764	SF	MEM	-	285.55	435.46
lmult0	1205	0.39	13.05	1.45	0.16	7.35
lmult2	3524	57.11	TIME	TIME	6.7	16.46
lmult4	6068	261.74	MEM	-	35.34	98.05
lmult6	8852	774.08	MEM	-	157.02	609.07
lmult8	11876	SF	MEM	-	297.32	704.08
qvar8	2272	0.62	MEM	17.67	2.95	36
qvar10	5621	2.21	MEM	234.97	55.54	156.44
qvar12	7334	6.2	MEM	-	36.8	74.49
qvar14	9312	SF	MEM	-	117.25	815.66
qvar16	6495	SF	MEM	-	51.33	117.31
c432	1114	131.06	TIME	7.22	0.24	7.6
c499	1869	TIME	TIME	100.41	0.8	4.59
c880	2589	TIME	TIME	320.96	5.54	38.91
c1355	3661	TIME	TIME	TIME	80.09	21.2
c1908	5095	TIME	MEM	TIME	58.01	165.99
c2670	6755	TIME	MEM	-	63.64	100.31
c3540	9325	TIME	MEM	-	242.02	786.33
u-bw.a	3290	7.81	TIME	249	209.03	178.18
u-bw.b	N/A	TIME	MEM	-	TIME	TIME
u-log.a	5783	TIME	MEM	TIME	59.65	179.3
u-log.b	6428	TIME	MEM	-	35.37	144.83
u-log.c	9506	TIME	MEM	-	383.65	731.87
u-rock.a	1691	13.29	TIME	41.29	206.56	6.26

Table 6.2: RMAXSAT vs. OPTSAT

CNF	OPTSAT		RMAXSAT		Bound DIFF	Speed UP
	#C	Run Time	#C	Run Time		
barrel3	941	0.335	941	0.02	0	16.75
barrel4	2034	2.26	2032	0.19	2	11.89
barrel5	5382	65.37	5380	2.11	2	30.98
barrel6	8930	257.48	8928	12.94	2	19.90
barrel7	13764	481.29	13763	17.63	1	27.30
lmult0	1205	2.36	1205	0.03	0	78.67
lmult2	3524	6.19	3523	0.1	1	61.90
lmult4	6068	49.74	6068	0.28	0	177.64
lmult6	8852	197.05	8852	7.2	0	27.37
lmult8	11876	787.3	11876	384.76	0	2.05
qvar8	2272	10.77	2272	0.18	0	59.83
qvar10	5621	49.18	5621	0.5	0	98.36
qvar12	7334	28.53	7334	0.95	0	30.03
qvar14	9312	188.09	9312	1.06	0	177.44
qvar16	6495	43.29	6495	1.2	0	36.08
c432	1114	2.41	1114	0.08	0	30.13
c499	1869	3.53	1869	0.3	0	11.77
c880	2589	14.43	2588	0.57	1	25.32
c1355	3661	10.88	3661	1.14	0	9.54
c1908	5095	61.04	5095	2.14	0	28.52
c2670	6755	43.14	6755	1.37	0	31.49
c3540	9325	353.27	9325	65.48	0	5.40
u-bw.a	3290	18.19	3285	0.27	5	67.37
u-bw.b	N/A	TIME	11480	1.14	N/A	N/A
u-log.a	5783	47.63	5782	0.22	1	216.50
u-log.b	6428	45.09	6428	0.23	0	196.04
u-log.c	9506	179.57	9506	0.39	0	460.44
u-rock.a	1691	1.59	1691	0.08	0	19.88
Average					0.56	72.54

Table 6.3: RMAXSAT vs. MSAT+

CNF	MSAT+		RMAXSAT		Bound DIFF	Speed UP
	#C	Run Time	#C	Run Time		
barrel3	941	0.04	941	0.02	0	2.23
barrel4	2034	0.04	2032	0.19	2	0.19
barrel5	5382	8.86	5380	2.11	2	4.20
barrel6	8930	27.45	8928	12.94	2	2.12
barrel7	13764	315.60	13763	17.63	1	17.90
lmult0	1205	0.05	1205	0.03	0	1.71
lmult2	3524	2.52	3523	0.1	1	25.20
lmult4	6068	17.93	6068	0.28	0	64.03
lmult6	8852	50.80	8852	7.2	0	7.06
lmult8	11876	332.46	11876	384.76	0	0.86
qvar8	2272	0.88	2272	0.18	0	4.90
qvar10	5621	17.46	5621	0.5	0	34.92
qvar12	7334	14.09	7334	0.95	0	14.84
qvar14	9312	27.04	9312	1.06	0	25.51
qvar16	6495	18.94	6495	1.2	0	15.78
c432	1114	0.08	1114	0.08	0	0.95
c499	1869	0.62	1869	0.3	0	2.05
c880	2589	2.05	2588	0.57	1	3.60
c1355	3661	41.10	3661	1.14	0	36.06
c1908	5095	21.33	5095	2.14	0	9.97
c2670	6755	27.37	6755	1.37	0	19.98
c3540	9325	108.73	9325	65.48	0	1.66
u-bw.a	3290	21.34	3285	0.27	5	79.03
u-bw.b	N/A	TIME	11480	1.14	N/A	N/A
u-log.a	5783	15.85	5782	0.22	1	72.03
u-log.b	6428	11.01	6428	0.23	0	47.88
u-log.c	9506	94.13	9506	0.39	0	241.36
u-rock.a	1691	52.46	1691	0.08	0	655.81
Average					0.56	51.55

Chapter 7

Summary of This Dissertation

As one of the critical backbone solvers, SAT solvers are currently drawing numerous attention in the research society, especially in the formal verification community where SAT is viewed as a major player in several key engines. It is envisioned that SAT will continue to play a critical role in the foreseeable future.

To continue the advances of the SAT solver, this dissertation is focused on leveraging the performance, reducing the cost of SAT solving and escalating the solver capability on optimized SAT problems. Three major topics about SAT techniques and their applications in formal verification were discussed in this dissertation.

In Chapter 1 we presented a brief introduction about why SAT is so important in formal verification. Then the motivation of this dissertation was discussed. Three major contributions of this dissertation were outlined: A hybrid SAT solution, double-layer conflict driven learning (DLSAT) and a novel MIN-ONE SAT algorithm named RelaxSAT. The other contributions include a MAX-SAT solver based on the RelaxSAT and its applications in bounded model checking. To help the readers to understand our proposed techniques, in Chapter 2 the basic knowledge about SAT and the solving techniques are described in detail.

Next, given the current research progress in SAT, we believe that besides the low-level tech-

niques like decision order, efficient BCP or better space pruning, systematic hybrid approaches should be studied to take advantage from various existing methodologies. In Chapter 3 we proposed a hybrid solution that can be adopted to many DPLL based solvers by combining the strength from tree based search like DPLL and local search algorithms like WALKSAT. Although it is not the first attempt to integrate these two major, yet different approaches on SAT solving, our approach is entirely different from the existing hybrids as it works iteratively under an abstraction-refinement framework thus to be more seamless and effective. The feedbacks between the DPLL and WALKSAT help to enhance the solution search for both of them. The experiments demonstrated that the hybrid solution is not only effective to boost the SAT solver performance but also highly portable to the existing SAT solvers.

Conflict-driven learning is one of the most critical techniques in the SAT solver and has been regarded as a groundbreaking milestone of SAT solving in the last decade. After careful investigation, we realized that there still exists much potential to improve conflict-driven learning. The existing conflict driven learning schemes treat all the conflict-include clauses equally, which hardly represent the reality that their importance and usefulness are discrepant. Some methods, such as clause database compaction, have been proposed to eliminate those useless (we call it supplementary) conflict-induced clauses periodically. However, they generally are passive, thus they can not alleviate the side effects on reducing the effectiveness of the resulting decision order. Our proposed double-layer conflict-driven learning identifies those conflict-induced clauses that are crucial to solve the problem, which we call primary clauses, proactively and store these primary conflict-induced clauses and the supplementary counterpart separately. To reduce the cost in BCP and storage, those supplementary conflict-induced clauses are represented compactly as Pseudo Boolean Constraints (PBC), where they were suppressed without losing all of their deductive power. It is a major contribution to save supplementary clauses as Pseudo boolean constraints since that the partial even complete space pruning power are kept, which is not the case for any previous attempts to remove useless conflict-induced clauses, like clause database compaction and clause deletion. Furthermore, we conducted extensive research on the assessment of the usefulness of the conflict-induced clause. These researches lead to the implementation of DLSAT which exhibited the value

of our proposed double-layer conflict driven learning by presenting the significant performance improvement it can bring.

For modern SAT solvers, they are not only limited to the traditional constraint solving, the optimized SAT problems are also their major targets due to the wide usage in many application domains. The most popular optimized SAT problems are MIN-ONE SAT and its sibling MAX-SAT. Both MIN-ONE SAT and MAX-SAT can be applied to problems in design verification and circuit testing. In MIN-ONE SAT, the solution of the formula is desired and an objective function based on the assignments is to be optimized. Different from the assignments optimization in MIN-ONE SAT, MAX-SAT optimizes the satisfiability of the formula where the number of the satisfied clauses is expected to be maximized. We proposed a conflict-driven relaxation based MIN-ONE SAT approximation algorithm (RelaxSAT) in Chapter 5. To the best of our knowledge this algorithm is the first approximation MIN-ONE SAT algorithm, which provides over an order of magnitude of speedup in performance and fairly tight bound. We believe that the RelaxSAT will have a significant impact on the applications of MIN-ONE SAT because it significantly reduces the cost of MIN-ONE SAT solving, thus many previously costly applications could become affordable.

Based on the proposed RelaxSAT, a MAX-SAT solver called RMAXSAT was described in Chapter 6. Thanks to the powerful underlying MIN-ONE SAT engine, RMAXSAT presents a considerable performance edge over other existing MAX-SAT solvers, including OPTSAT and MSAT+. In Chapter 6 we also discussed and forecasted the potential of deploying RMAXSAT in bounded model checking.

With the three major contributions: 1) Hybrid SAT solution, 2) Double-layer conflict driven learning and 3) A fast approximation algorithm for MIN-ONE SAT and MAX-SAT, this dissertation explores both the research depth and width in SAT techniques, especially in the formal verification area. Because we were inspired by some connected underlying principles in our research, it turned out that our contributions share the similarities between them: they were proposed under common guidelines such as abstraction and refinement, hybrid solution and conflict-driven relaxation. We believe the research that has been presented in this dissertation serves a substantial

foundation to many ongoing and future researches, both to the SAT and EDA communities.

Bibliography

- [1] IBM Research, http://www.haifa.ibm.com/projects/verification/RB_Homepage/bmcbenchmarks.html, 2007.
- [2] SAT 2002 Competition Benchmarks, <http://www.satlib.org/>, 2002.
- [3] Minisat+, <http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/MiniSat+.html>, 2005.
- [4] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on sat-solvers. In *TACAS*, pages 411–425, 2000.
- [5] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. Pbs: A backtrack search pseudo-boolean solver. In *Proceedings of International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 46–353, 2002.
- [6] T. Alsinet, F. Manyk, and J. Planes. Improved branch and bound algorithms for max-sat. In *Proceedings of International Conference on the Theory and Applications of Satisfiability Testing (SAT)*, pages 408–415, 2003.
- [7] T. Alsinet, F. Manyà, and J. Planes. A max-sat solver with lazy data structures. In *Proceedings of Advances in Artificial Intelligence - IBERAMIA*, pages 334–342, 2004.
- [8] J. Argelich and F. Manyà. Partial max-sat solvers with clause learning. In *Proceedings of International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 28–40, 2007.

- [9] T. Asano, T. Ono, and T. Hirata. Approximation algorithms for the maximum satisfiability problem. *Nordic J. of Computing*, 3(4):388–404, 1996. ISSN 1236-6064.
- [10] A. Aziz, F. Balarin, S.-T. Cheng, R. Hojati, T. Kam, S. C. Krishnan, R. K. Ranjan, T. R. Shiple, V. Singhal, S. Tasiran, H.-Y. Wang, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Hsis: A bdd-based environment for formal verification. In *Proceedings of Design Automation Conference (DAC)*, pages 454–459, 1994.
- [11] O. Bailleux and P. Marquis. Distance-sat: Complexity and algorithms. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 642–647, 1999.
- [12] S. Barner, D. Geist, and A. Gringauze. Symbolic localization reduction with reconstruction layering and backtracking. In *Proceedings of Conference on Computer-Aided Verification*, pages 65–77, 2002.
- [13] P. Barth. A davis-putnam enumeration algorithm for linear pseudo-boolean optimization. Technical Report MPI-I-95-2-2003, Max Plank Institute for Computer Science, 1995.
- [14] R. J. J. Bayardo and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 203–208, Providence, Rhode Island, 1997.
- [15] A. Bertoni, P. Campadelli, M. Carpentieri, and G. Grossi. A genetic model: Analysis and application to maxsat. *Evolutionary Computation*, 8(3):291–309, 2000. ISSN 1063-6560. doi: <http://dx.doi.org/10.1162/106365600750078790>.
- [16] M. L. Bonet, J. Levy, and F. Manyà. A complete calculus for max-sat. In *Proceedings of International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 240–251, 2006.
- [17] M. L. Bonet, J. Levy, and F. Manyà. Resolution for max-sat. *Artif. Intell.*, 171(8-9):606–618, 2007. ISSN 0004-3702. doi: <http://dx.doi.org/10.1016/j.artint.2007.03.001>.

- [18] B. Borchers and J. Furman. A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. *Journal of Combinatorial Optimization*, 2:299–306, 1999.
- [19] E. Boros and P. Hammer. Pseudo-boolean optimization, 2002.
- [20] R. Bryant. Graph-based algorithms for boolean function manipulation. In *IEEE Transactions on Computers*, volume c-35, pages 677–691, August 1986.
- [21] R. E. Bryant. Binary decision diagrams and beyond: enabling technologies for formal verification. In *Proceedings of Internatioanl Computer-Aided Design Conference (ICCAD)*, pages 236–243, 1995.
- [22] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of Design Automation Conference (DAC)*, pages 46–51, 1990.
- [23] B. Cha, K. Iwama, Y. Kambayashi, and S. Miyazaki. Local search algorithms for partial MAXSAT. In *Proceedings of National Conference on Artificial Intelligence (AAAI/IAAI)*, pages 263–268, 1997.
- [24] D. Chai and A. Kuehlmann. A fast pseudo-boolean constraint solver. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 24(3):305–317, 2005.
- [25] C. M. Chang. Verification: port in a storm. <http://www.eetimes.com/story/OEG20010725S0044>, 2001.
- [26] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Pres, 2000.
- [27] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Workshop of Logic of Programs*, pages 52–71, 1981.
- [28] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

- [29] M. Davis, G. Logemann, and D. W. Loveland. Machine program for theorem proving. In *Communications of the ACM*, volume 5, pages 394–397, 1962.
- [30] N. Eén and N. Sörensson. An extensible sat-solver. In *Proceedings of International Conference on the Theory and Applications of Satisfiability Testing*, pages 502–518, 2003.
- [31] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into sat. In *JSAT*, volume 2, pages 1–26, 2006.
- [32] L. Fang and M. S. Hsiao. A new hybrid solution to boost sat solver performance. In *Proceedings of the Conference on Design, Automation & Test in Europe(DATE)*, pages 1307–1313, 2007.
- [33] L. Fang and M. S. Hsiao. A fast approximation algorithm for min-one sat. In *Proceedings of the Conference on Design, Automation & Test in Europe(DATE)*, pages 1087–1090, 2008.
- [34] R. W. Fang H. Complete local search for propositional satisfiability. In *Proc. of 19th National Conference on Artificial Intelligence*, pages 161–166, 2004.
- [35] 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.pd>, 2007.
- [36] Z. Fu and S. Malik. On solving the partial max-sat problem. *Theory and Applications of Satisfiability Testing - SAT 2006*, 4121(4):252–265, 2006. ISSN 0302-9743.
- [37] A. S. Fukunaga. Efficient implementations of sat local search. In *Proceedings of International Conference on the Theory and Applications of Satisfiability Testing*, 2004.
- [38] E. Giunchiglia and M. Maratea. Solving optimization problems with dll. In *Proceedings of European Conference on Artificial Intelligence*, pages 377–381, 2006.
- [39] E. I. Goldberg, M. Prasad, and R. K. Brayton. Using sat for combinational equivalence checking. In *Proceedings of Design, Automation & Test in Europe(DATE)*, pages 114–121, 2001.

- [40] A. Gupta, Z. Yang, P. Ashar, and A. Gupta. Sat-based image computation with application in reachability analysis. In *Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 354–371, London, UK, 2000. Springer-Verlag. ISBN 3-540-41219-0.
- [41] A. Gupta, Z. Yang, P. Ashar, and A. Gupta. Sat-based image computation with application in reachability analysis. In *Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 354–371, 2000.
- [42] A. Gupta, Z. Yang, P. Ashar, L. Zhang, and S. Malik. Partition-based decision heuristics for image computation using sat and bdds. In *Proceedings of International Computer-Aided Design Conference (ICCAD)*, pages 286–292, Piscataway, NJ, USA, 2001. IEEE Press. ISBN 0-7803-7249-2.
- [43] 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. ISBN 3-540-44120-4.
- [44] A. J. Hu, D. L. Dill, A. J. Drexler, and C. H. Yang. Higher-level specification and verification with bdds. In *CAV*, pages 677–691, 1992.
- [45] K. Lieberherr. *Information Condensation of Models in the Propositional Calculus and the $P=NP$ Problem*. PhD thesis, ETH Zurich, 1977. 145 pages, in German.
- [46] H.-J. Kang and I.-C. Park. Sat-based unbounded symbolic model checking. In *Proceedings of Design Automation Conference (DAC)*, pages 840–843, 2003.
- [47] H. Kriplani, F. Najm, and I. Hajj. *Worst case voltage drops in power and ground buses of CMOS VLSI circuits*. PhD thesis, University of Illinois at Urbana-Champaign, 1994.
- [48] J. Larrosa, F. Heras, and S. de Givry. A logical approach to efficient max-sat solving. *Artif. Intell.*, 172(2-3):204–233, 2008. ISSN 0004-3702. doi: <http://dx.doi.org/10.1016/j.artint.2007.05.006>.

- [49] B. Li, M. S. Hsiao, and S. Sheng. A novel sat all-solutions solver for efficient preimage computation. In *Proceedings of the Conference on Design, Automation & Test in Europe(DATE)*, page 10272, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2085-5-1.
- [50] B. Li, L. Fang, and M. S. Hsiao. Efficient power droop aware delay fault testing. In *International Test Conference*, pages 1–10, October 2009.
- [51] A. Mali. Hybrid propositional encodings of planning. page 972, 1999.
- [52] A. D. Mali. On quantified weighted max-sat. *Decision Support Systems*, 40(2):257–268, August 2005.
- [53] P. Manolios and S. K. Srinivasan. A parameterized benchmark suite of hard pipelined-machine-verification problems. In *CHARME*, pages 363–366, 2005.
- [54] J. P. Marques-Silva and K. A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.
- [55] D. Martin and P. Hillary. Computing procedure for quantification theory. In *Journal of the ACM*, volume 7, pages 201–215, 1960.
- [56] 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. ISSN 1012-2443.
- [57] K. L. McMillan. Applying sat methods in unbounded symbolic model checking. In *CAV*, pages 250–264, 2002.
- [58] K. L. McMillan. Interpolation and sat-based model checking. In *CAV*, pages 1–13, 2003.
- [59] M. E. Menai and M. Batouche. An effective heuristic algorithm for the maximum satisfiability problem. *Applied Intelligence*, 24(3):227–239, 2006. ISSN 0924-669X. doi: <http://dx.doi.org/10.1007/s10489-006-8514-7>.
- [60] A. Meyer. *Principles of Functional Verification*. Newnes, October 1990.

- [61] M. Mneimneh and K. Sakallah. Sat-based sequential depth computation. In *ASPDAC: Proceedings of the 2003 conference on Asia South Pacific design automation*, pages 87–92, New York, NY, USA, 2003. ACM. ISBN 0-7803-7660-9. doi: <http://doi.acm.org/10.1145/1119772.1119790>.
- [62] 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)*, 2001.
- [63] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of International Symposium on Programming*, pages 337–350, 1981.
- [64] P. Rashinkar, P. Paterson, and L. Singh. *System-on-a-Chip Verification - Methodology and Techniques*. Springer, 2000. ISBN 0792372794.
- [65] M. Resende, L. Pitsoulis, and P. Pardalos. Approximate solution of weighted max-sat problems using grasp, 1996.
- [66] 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 National Conference on Artificial Intelligence (AAAI)*, pages 440–446, Menlo Park, California, 1992. AAAI Press.
- [67] B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 337–343, Seattle, 1994.
- [68] M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a sat solver. In *Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD), Lecture Notes in Computer Science*, pages 108–125, 2000.
- [69] H. M. Sheini and K. A. Sakallah. Pueblo: A modern pseudo-boolean sat solver. In *Proceedings of Design, Automation & Test in Europe (DATE)*, pages 684–685, 2005.
- [70] S. Sheng and M. Hsiao. Efficient preimage computation using a novel success-driven atpg. In *Proceedings of Design, Automation & Test in Europe (DATE)*, pages 822–827, 2003.

- [71] O. Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. *Lecture Notes in Computer Science*, 2144:58–70, 2001.
- [72] J. P. M. Silva and T. Glass. Combinational equivalence checking using satisfiability and recursive learning. In *Proceedings of Design, Automation & Test in Europe (DATE)*, pages 145–149, 1999.
- [73] J. Thornton, S. Bain, A. Sattar, and D. N. Pham. A two level local search for max-sat problems with hard and soft constraints. In *AI '02: Proceedings of the 15th Australian Joint Conference on Artificial Intelligence*, pages 603–614, London, UK, 2002. Springer-Verlag. ISBN 3-540-00197-2.
- [74] M. Velev. http://www.ece.cmu.edu/~mvelev/sat_benchmarks.html, 2007.
- [75] J. Whitemore, J. Kim, and K. A. Sakallah. Satire: A new incremental satisfiability engine. In *Proceedings of Design Automation Conference (DAC)*, pages 542–545, 2001.
- [76] Z. Xing and W. Zhang. Maxsolver: an efficient exact algorithm for (weighted) maximum satisfiability. *Artificial Intelligence*, 164(1-2):47–80, 2005. ISSN 0004-3702. doi: <http://dx.doi.org/10.1016/j.artint.2005.01.004>.
- [77] M. Yoeli. *Formal verification of hardware design*. Los Alamos, Calif. : IEEE Computer Society Press, 1990. ISBN 0818690178.
- [78] L. Zhang and S. Malik. Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of Design, Automation & Test in Europe (DATE)*, pages 10880–10885, 2003.
- [79] L. Zhang and S. Malik. Conflict driven learning in a quantified boolean satisfiability solver. In *Proceedings of Internatioanl Computer-Aided Design Conference (ICCAD)*, pages 442–449, 2002.

- [80] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formulas. In *Proceedings of International Conference on the Theory and Applications of Satisfiability Testing*, May 2003.
- [81] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *Proceedings of Internatioanl Computer-Aided Design Conference (ICCAD)*, pages 279–285, 2001.