On Solving The Partial MAX-SAT Problem*

Zhaohui Fu** and Sharad Malik

Department of Electrical Engineering
Princeton University
Princeton, NJ 08544, USA
{zfu,sharad}@Princeton.EDU

Abstract. Boolean Satisfiability (SAT) has seen many successful applications in various fields such as Electronic Design Automation and Artificial Intelligence. However, in some cases, it may be required/preferable to use variations of the general SAT problem. In this paper, we consider one important variation, the Partial MAX-SAT problem. Unlike SAT, Partial MAX-SAT has certain constraints (clauses) that are marked as relaxable and the rest are hard, i.e. non-relaxable. The objective is to find a variable assignment that satisfies all non-relaxable clauses together with the maximum number of relaxable ones. We have implemented two solvers for the Partial MAX-SAT problem using a contemporary SAT solver, zChaff. The first approach is a novel diagnosis based algorithm; it iteratively analyzes the UNSAT core of the current SAT instance and eliminates the core through a modification of the problem instance by adding relaxation variables. The second approach is encoding based; it constructs an efficient auxiliary counter that constrains the number of relaxed clauses and supports binary search or linear scan for the optimal solution. Both solvers are complete as they guarantee the optimality of the solution. We discuss the relative strengths and thus applicability of the two solvers for different solution scenarios. Further, we show how both techniques benefit from the persistent learning techniques of incremental SAT. Experiments using practical instances of this problem show significant improvements over the best known solvers.

1 Introduction

In the last decade Boolean Satisfiability (SAT) has seen many great advances, including non-chronological backtracking, conflict driven clause learning, efficient Boolean Constraint Propagation (BCP) and UNSAT core generation. As a consequence, many applications have been able to successfully use SAT as a decision procedure to determine if a specific instance is SAT or UNSAT. However, there are many other variations of the SAT problem that go beyond this decision procedure use of SAT solvers. For example, the MAX-SAT Problem [8] seeks the maximum number of clauses that can be satisfied. This paper examines a generalization of this problem referred to as Partial MAX-SAT [2,15].

* This research is supported by a grant from the Air Force Research Laboratory.

** Z. Fu is on leave from the Department of Computer Science, National University of Singapore, Singapore 117543.

Partial MAX-SAT [2,15] (PM-SAT) sits between the classic SAT problem and MAX-SAT. While the classic SAT problem requires *all* clauses to be satisfied, PM-SAT relaxes this requirement by having certain clauses marked as *relaxable* or *soft* and others to be *non-relaxable* or *hard*. Given n relaxable clauses, the objective is to find an assignment that satisfies all non-relaxable clauses together with the maximum number of relaxable clauses (i.e. a minimum number k of these clauses get *relaxed*). PM-SAT can thus be used in various optimization tasks, e.g. multiple property checking, FPGA routing, university course scheduling, etc. In these scenarios, simply determining that an instance is UNSAT is not enough. We are interested in obtaining a best way to make the instance satisfiable allowing for *some* clauses to be unsatisfied.

The difference between PM-SAT and MAX-SAT [8] is that every clause in MAX-SAT can be relaxed, which clearly makes MAX-SAT a special case of PM-SAT. Though decision versions of both problems are NP-Complete [5], PM-SAT is clearly more versatile.

1.1 Previous Work

PM-SAT was first defined by Miyazaki *et al.* [15] during their work on optimization of database queries in 1996. In the same year, Kautz *et al.* [10] proposed the first heuristic algorithm based on local search for solving this problem. Later in 1997 Cha *et al.* [2] proposed another local search technique to solve the PM-SAT problem in the the context of university course scheduling.

In 2005, Li used two MinCostSat solvers, *eclipse-stoc* [12] and *wpack* [12], for the transformed PM-SAT problem in FPGA routing. MinCostSat is a SAT problem which minimizes the cost of the satisfying assignment. For example, assigning a variable to be true usually incurs a positive cost while assigning it to be false incurs no cost. The objective is to find a satisfying assignment with minimum total cost. By inserting a slack variable [12] to each of the relaxable clauses, Li transforms the PM-SAT problem into a MinCostSat problem with each slack variable having a unit cost. *eclipse-stoc* is a general purpose MinCostSat solver and *wpack* is specialized for FPGA routing benchmarks. Li demonstrated some impressive results using *wpack* in his thesis. However, both *eclipse-stoc* and *wpack* are based on local search techniques and hence are not complete solvers, i.e. the solver provides no guarantee on the optimality of the solution.

Argelich and Manyà uses a branch and bound approach for the over constrained MAX-SAT problems [1]. However, as we will show in Section 4, branch and bound based algorithms, including *bsolo*, do not work well on the PM-SAT problem.

1.2 Our Contribution

In this paper, we propose two practically efficient approaches to solve the PM-SAT problem optimally. Both approaches use the state-of-the-art SAT solver zChaff [16] with certain extensions.

1. Diagnosis Based. The first approach is based on the ability of SAT solvers to provide an UNSAT core [21] for unsatisfiable instances. This core is a subset of original clauses that are unsatisfiable by themselves and in some sense can be considered

to be the “cause” of the unsatisfiability. This core is generated as a byproduct of the proof of the unsatisfiability. The UNSAT core is analyzed and each relaxable clause appearing in the core is augmented with a distinct relaxation variable. Additional clauses are added to the original SAT instance to ensure the *one-hot* property of these relaxation variables. This augmentation essentially *eliminates* this core from the SAT instance. The procedure continues until the SAT instance is satisfiable. We give a proof of the optimality of the final solution using these relaxation variables and the one-hot property.

2. Encoding Based. The second approach constructs an efficient auxiliary logic counter, i.e. an adder and comparator, to constrain the number of clauses that can be relaxed simultaneously. It then uses either *binary search* or *linear scan* techniques to find the minimum number of k (out of n) clauses that need to be relaxed. The logic counter is carefully designed such that maximum amount of learned information can be re-used across different invocations of the decision procedure.

2 Diagnosis Based Approach: Iterative UNSAT Core Elimination

Being the best solver in the Certified UNSAT Track of SAT 2005 Competition, zChaff is very efficient in generating UNSAT cores. Our diagnosis based approach takes full advantage of this feature. It iteratively identifies the reason of the unsatisfiability of the instance, i.e. the UNSAT core [21], and uses relaxation variables to eliminate these UNSAT cores one by one until the instance becomes satisfiable.

Definition 1. *An unsatisfiable core is a subset of the* original *CNF clauses that are unsatisfiable by themselves.*

Modern SAT solvers provide the UNSAT core as a byproduct of the proof of unsatisfiability [21].

2.1 The Optimal Algorithm with Proof

The diagnosis based approach is illustrated in Algorithm 1. We use CNF to represent the original SAT instance and $V(CNF)$ is the set of all Boolean variables and $C(CNF)$ is the set of all clauses. An UNSAT core UC is a set of clauses, i.e. $UC \subseteq C(CNF)$. A clause $c \in C(CNF)$ consists of a set of literals. A literal l is just a Boolean variable, $v \in V(CNF)$, with positive or negative phase, i.e. $l = v$ or $l = v'$.

Given an UNSAT core UC, for each relaxable clause $c \in UC$, a distinct relaxation variable is added to this clause, i.e. c is replaced by $c \cup \{v\}$. Setting this variable to true makes the associated clause satisfied (and hence relaxed). An UNSAT core is said to be *eliminated* when at least one of its clauses is satisfied (relaxed) by a relaxation variable setting to be true.

Let S be the set of relaxation variables from UNSAT core UC, the one-hot constraint over a set S of Boolean variables requires that one and only one of the variables in S is assigned to be true and the other $|S|-1$ variables must be false. The number of clauses added due to the one-hot constraint is $\frac{|S|\times(|S|-1)}{2}+1$. For example, with $S=\{a,b,c\}$ and the one-hot constraint clauses are $(a'+b')(b'+c')(a'+c')(a+b+c)$.

Algorithm 1 Iterative UNSAT Core Elimination

1: $\mathcal{S} := \emptyset$
2: **while** SAT solver returns UNSATISFIABLE **do**
3: Let UC be the UNSAT core provided by the SAT solver
4: $S := \emptyset$
5: **for all** Clause $c \in UC$ **do**
6: **if** c is relaxable **then**
7: Allocate a new relaxation variable v
8: $c := c \cup \{v\}$
9: $S := S \cup \{v\}$
10: **end if**
11: **end for**
12: **if** $S = \emptyset$ **then**
13: Return CNF UNSATISFIABLE
14: **else**
15: Add clauses enforcing the *One-Hot* constraint for S to the SAT solver
16: $\mathcal{S} := \mathcal{S} \cup S$
17: **end if**
18: **end while**
19: $R := \{v \mid v \in \mathcal{S}, v = 1\}$; $k := |R|$
20: Return Satisfying Assignment, k, R.

One relaxable clause is relaxed during each UNSAT iteration of the `while` loop in Algorithm 1. The algorithm stops after exactly k iterations, where k is the minimum number of clauses to be relaxed. R is the subset of S consisting of all relaxation variables set to 1, i.e. the corresponding clauses are relaxed.

Theorem 1. *Algorithm 1 finds the minimum number of clauses to be relaxed.*

Proof. Consider the interesting case where the original problem is always satisfiable with relaxation of certain relaxable clauses (Otherwise Algorithm 1 returns unsatisfiable in Line 12). Suppose Algorithm 1 stops after exactly k iterations, i.e. relaxing k clauses. Clearly, Algorithm 1 has encountered a total number of k UNSAT cores (one in each iteration), which we denote them by a set $U, |U| = k$. Note that original problem instance contains at least k UNSAT cores, even though each iteration starts with a new modified problem instance. Now suppose that there exists some optimal solution that relaxes a set $M, |M| < k$, clauses to make the original problem satisfiable. Obviously, relaxing all clauses in M eliminates all the UNSAT cores in U. However, since $|M| < k = |U|$ and by the Pigeon Hole Principle there must be at least one clause $c \in M$ those relaxation eliminates two or more UNSAT cores $(u_1, u_2, \ldots) \in U$ and $c \in u_1, c \in u_2$. Without the loss of generality, let us assume that Algorithm 1 encounters u_1 first in the `while` loop. So every clause including c in u_1 is added with a relaxation variable. Let v be the relaxation variable added to c. Now there exists an assignment that can eliminate both u_1 and u_2 by setting v to be true. Due to the completeness of our SAT solver, the UNSAT core u_2 should never be encountered, which leads to a contradiction. This contradiction is caused by the assumption that $|M| < k = |U|$. Hence we have to conclude that $|M| = k = |U|$, i.e. each relaxed clause in M can eliminate at most one UNSAT core in U. □

It is worth mentioning that the UNSAT core extraction is not compulsory. One could add a relaxation variable to each relaxable clause and require this batch of relaxation variables to be one-hot for every iteration in which the problem remains unsatisfiable. This naive approach is still capable of finding the minimum number of clauses to be relaxed. However, recall that the one-hot constraint requires $O(|S|^2)$ additional clauses where S is the set of relaxable clauses in the UNSAT core. Therefore it is impractical to enforce the one-hot constraint on the relaxation variables for all relaxable clauses. For example, the PM-SAT instance might have 100 relaxation clauses while only 3 appear in the UNSAT core. The naive approach adds $\frac{100\times99}{2}+1=4951$ clauses while the diagnosis based approach adds only $\frac{3\times2)}{2}+1=4$ clauses. The diagnosis based approach exploits the availability of the UNSAT core to keep the number of relaxation variables and one-hot constraint clauses small.

2.2 An Illustrative Example

It is worth mentioning that Algorithm 1 does not require the UNSAT core UC to be minimal. Furthermore, the UNSAT cores encountered by Algorithm 1 need not be disjoint. The following example shows a simple CNF formula that contains two overlapping cores. Suppose we have four Boolean variables x_1, x_2, x_3 and x_4. Relaxable clauses are shown with square brackets and $\odot$ denotes the *resolution* operator.

$$(x_1'+x_2')(x_1'+x_3)(x_1'+x_3')(x_2'+x_4)(x_2'+x_4')[x_1][x_2]$$

This CNF formula is unsatisfiable since $(x_1'+x_2')[x_1][x_2]$ form an UNSAT core because

$$(x_1'+x_2')\odot[x_1]\odot[x_2]=(x_2')\odot[x_2]=()$$

Note that whether a clause is relaxable or not does not affect the resolution. Recall that a UNSAT core is a set of original clauses that are unsatisfiable and they resolve to an empty clause $()$, which can never be satisfied. The only relaxable clauses in this core are $[x_1][x_2]$. So in the first iteration we add two distinct relaxation variables r_1 and r_2 to each of them respectively and enforce r_1 and r_2 to be one-hot. The resulting CNF formula is

$$(x_1'+x_2')(x_1'+x_3)(x_1'+x_3')(x_2'+x_4)(x_2'+x_4')[x_1+r_1][x_2+r_2](r_1'+r_2')(r_1+r_2)$$

Note that clauses due to the one-hot constraint are not relaxable. However, the relaxable clauses are still marked as relaxable even after inserting relaxation variables. This is because, as we will show, one relaxation variable may not be enough to make the instance satisfiable. The current CNF formula is still unsatisfiable as

$$\begin{aligned}(x_1'+x_3)\odot(x_1'+x_3')\odot[x_1+r_1]\odot(r_1'+r_2')\odot[x_2+r_2] &= (x_2)\\ (x_2)\odot(x_2'+x_4') &= (x_4')\\ (x_2)\odot(x_2'+x_4) &= (x_4)\\ (x_4')\odot(x_4) &= ()\end{aligned}$$

So in the second iteration, we add another two relaxation variables r_3 and r_4 to the relaxable clauses $[x_1 + r_1][x_2 + r_2]$ in the core. Together with clauses due to the one-hot constraint of r_3 and r_4, the CNF formula becomes

$$(x_1' + x_2')(x_1' + x_3)(x_1' + x_3')(x_2' + x_4)(x_2' + x_4')[x_1 + r_1 + r_3][x_2 + r_2 + r_4]$$
$$(r_1' + r_2')(r_1 + r_2)(r_3' + r_4')(r_3 + r_4)$$

This formula is satisfi able with the following assignment

$$\begin{array}{llll} x_1 = 0, & x_2 = 0, & x_3 = 1, & x_4 = 1, \\ r_1 = 1, & r_2 = 0, & r_3 = 0, & r_4 = 1. \end{array}$$

Based on the above satisfying assignment, both $[x_1][x_2]$ should be relaxed to make the problem satisfi able, i.e. $k = n = 2$. Note that there is no constraint among the relaxation variables added in different iterations and the one-hot constraint only applies to all relaxation variables added due to the same UNSAT core *in one iteration*. The number of the relaxation variables needed only depends on the number of relaxation clauses in the current UNSAT core[1], and not the total number of relaxation clauses in the entire SAT instance. In Section 4 we will see some cases where the total number of relaxable clauses is large and our diagnosis based approach still performs well on these cases.

The iterative core elimination requires the SAT solver to be able to provide the UNSAT core (or proof) as part of answering UNSAT. This feature does incur some overhead. For example, the SAT solver needs to record the resolution trace for each learned clause[2]. Even when a learned clause is deleted, which happens very frequently in most state-of-the-art SAT solvers, the resolution trace for that particular learned clause cannot be deleted because it might be used to resolve other learned clauses that are not yet deleted. In case of an unsatisfi able instance, we need all the resolution information so that we could trace back from the conflict to the original clauses, which then form the UNSAT core. Recording the resolution trace not only slows down the search speed, but also uses a large amount of memory, which could otherwise be used for learned clauses.

3 Encoding Based Approach: Constructing An Auxiliary Counter

With highly optimized state-of-the-art SAT solvers like zChaff [16], Berkmin [6], Siege [17] and MiniSat [4] , the most straightforward way is to translate the PM-SAT problem directly into a SAT instance. Such an implementation is likely to be effi cient since the translated SAT instance takes advantages of all the sophisticated techniques used in a contemporary SAT solver. Furthermore, this approach requires very little or no modifi - cation to the SAT solver itself and hence could continuously benefi t from the advances in SAT.

[1] Recall that we assume there must exist at least one relaxation clause in every core since otherwise the problem is unsatisfiable even if we relax all relaxable clauses.

[2] Each learned clause is the result of a series of resolutions of other clauses, both learned or original. But ultimately, each learned conflict clause is the result of a series of resolutions using only the original clauses.

However, conventional SAT solvers do not support integer arithmetic, which is necessary in PM-SAT for expressing the constraint of $\leq k$ *clauses left unsatisfied*. We use an auxiliary logic counter [11] to represent this $\leq k$ condition, whose output is a Boolean variable and the entire counter could then be translated into CNF in a straightforward way. There are various ways of constructing such an an auxiliary logic counter [11]. Xu considers four types of these logic counters, namely chain counter, hierarchical tree counter, routing counter and sorting counter in her work on *subSAT* [20]. *subSAT* is a MAX-SAT problem but with the assumption that $k \ll n$, where n here is the total number of clauses. In other words, the problem becomes satisfi able with very small number (usually $k < 5$) of clauses removed (relaxed). In the *subSAT* implementation, one *mask* variable (which is equivalent to our relaxation variable) is added to each of the n clauses and they constrain that only $\leq k$ mask variables can be true by using one of the four logic counters as mentioned above. The chain counter method creates a $\lceil \lg k+1 \rceil$ bit adder for each clause and concatenates them together. The fi nal output from the last adder is constrained to be $\leq k$. The hierarchical tree counter creates a tree using $\lceil \lg k+1 \rceil$ bit adders as internal nodes that sum up all n mask variables and gives a $\lceil \lg k+1 \rceil$ bit output at the root of the tree. The routing counter implements k k-to-n decoders with k inputs all set to be 1. The sorting counter uses a sorting circuit with k `max` operators (range from n bit to $n-k+1$ bit) to move the 1s to one side of the output and then checks the kth bit of the output. Xu states that the fi rst two counters (chain and tree counters) are more effi cient than the others in terms of the amount of additional logic.

3.1 An Efficient Hierarchical Tree Adder

The most signifi cant differences between our proposed encoding based approach and the *subSAT* approach are that our hierarchical tree adder is independent of k and we do not assume $k \ll n$. In addition to the linear scan for minimum k, we also use a binary search on $[0,n]$ for the minimum k (*subSat* only uses linear scan due to their assumption of $k \ll n$). We design our tree adder to be independent of the value k for two obvious reasons. First, we only need to construct the adder once at the beginning and re-use it during each iteration of the binary search, as compared to constructing the adder $\lg n$ times for binary search and k times for linear scan. Second, using the idea of incremental SAT [18], all clauses associated with the adder can be kept intact since they are always consistent with the problem. Maintaining the learned information is very important to the performance of most contemporary SAT solvers. Unfortunately, all the above 4 types of auxiliary counters proposed by Xu are dependent on k, particularly for the routing and sorting counters.

We propose a hierarchical tree adder that is independent of k using elementary adders, e.g. half adder and full adders. Figure 1 gives an example of such an adder with $n = 9$. It can be shown that the total number of additional 2-input logic gates is $\leq 5n$ as follows. Consider starting with $n \geq 3$ bit input, we use a full adder to sum up 3 bits while returning a sum bit and a carry bit [9]. The sum bit needs to be added with the other $n-3$ bits left and the carry bit will only be used in next level. So each full adder reduces the number of inputs left by 2 and $\lfloor \frac{n}{2} \rfloor$ full adders are suffi cient for the fi rst level. Note the last sum bit becomes the least signifi cant bit of the fi nal sum. In the

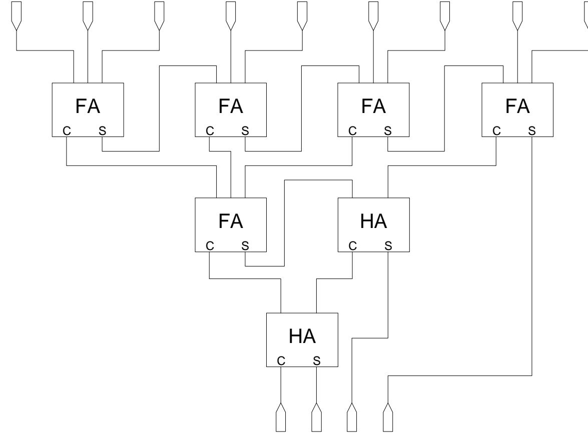

Fig. 1. An efficient hierarchical tree adder that sums the number of 1s from the $n = 9$ bit input (top) and gives a 4 bit binary value (bottom). The first level uses 4 full adders (FA); the second level needs 1 full adder and 1 half adder (HA); the third level just needs 1 half adder. S and C are the sum and carry bits of the adder respectively.

second level, we consider all the carry bits from the previous level and there are at most $\lfloor \frac{n}{2} \rfloor$ of them. Similar results extends to the third level and so on. So the total number of full adders is:

$$\lfloor \frac{n}{2} \rfloor + \lfloor \frac{n}{4} \rfloor + \lfloor \frac{n}{8} \rfloor + \ldots + 1 \leq n$$

Each full adder requires 5 2-input logic gates (2 AND, 2XOR and 1OR gate), which gives the total number of additional logic gates $\leq 5n$. Note that we can sometimes replace a full adder by a half adder due to simplification by constant value (0), as shown in the second and third levels in Figure 1.

There is an important distinction between our hierarchical tree adder in Figure 1 and the one used by Xu [20]. Instead of using full/half adders as internal nodes of the tree, Xu uses a $\lceil \lg k + 1 \rceil$ bit adder for each of the internal nodes, which introduces a large amount of redundancy with relatively large k. For example, the first level inputs to the adder are at most 1 and in a binary representation of $\lceil \lg k + 1 \rceil$ bits, at least $\lceil \lg k + 1 \rceil - 1$ bits are just 0s. Our hierarchical tree adder is free of such redundancy due to the judicious use of full/half adders.

The hierarchical tree adder outputs a $\lceil \lg n + 1 \rceil$ bit binary value, which is then compared against a given value k using a logic comparator that outputs true if and only if the sum is less than or equal to k. Note that this logic comparator is dependent on k for efficiency reasons[3]. This hierarchical tree adder with comparator provides us an efficient platform for searching the minimum k. Generally binary search has advantages over linear scan on the benchmarks with $k > \lg n$.

[3] The resulting circuit is equivalent to performing the constant propagation on a general logic comparator with any given k.

It is worth mentioning that when this logic counter (adder with comparator) are translated into CNF clauses, the hierarchical tree adder generates many more clauses than the comparator does. In general, the number of CNF clauses from the adder is $O(n)$ while from the comparator is $O(\lg n)$. For each iteration during the binary search or linear scan, only $O(\lg n)$ original CNF clauses with related learned clauses need to be changed. The remaining clauses include both original and learned clauses corresponding to the original problem instance and the adder. The learned clauses capture the logic relationship among the Boolean variables used in the problem instance and the adder and they cannot be learned without the adder.

One disadvantage of using an auxiliary counter is the introduction of a large number of XOR gates. Each full adder consists of two XOR gates and the entire counter results in $2n$ XOR gates. Though the number of additional logic gates is only linear in n, the situation could get worse when many XOR gates are chained together. For example, the least significant bit of the sum comes from an XOR chain of length $\lceil \lg n \rceil$. XOR chains are well known to cause poor performance of SAT solvers. One main reason is that unlike AND/OR gates, Boolean constraint propagation over XOR gates is very limited. This complication makes this approach no longer efficient for solving problems with very large n. It is worth mentioning that a large n does not necessarily imply a large k though obviously $k \leq n$.

3.2 A Discussion on Incremental SAT

Incremental SAT was first formalized by Strichman [18]. It is the process of solving a series of SAT instances $\varphi_1, \varphi_2, \ldots, \varphi_n$. The consecutive SAT instances, φ_i and φ_{i+1}, are *similar*, i.e. only a small number of clauses (and variables) are different. Given the solution of φ_i, we could solve φ_{i+1} *incrementally* by only updating the different clauses while keeping most learned clauses in φ_i, which are still consistent with φ_{i+1}, intact. Maintaining the maximum amount of the learned clauses, i.e. recording the most visited search space, is a great advantage than starting from scratch each time.

A key issue in the implementation of incremental SAT is the efficient updating from instance φ_i to φ_{i+1}, which usually includes both deletion and addition of original clauses. Addition of new original clauses is trivial. However, deletion of original clauses implies the additional deletion of all learned clauses related to the deleted original clauses in order to maintain the integrity of the clauses database. This deletion can be performed efficiently with the use of group IDs. A group ID indicates a particular group, to which the clause belongs. The group IDs of a learned clause is the *union* of all the group IDs from the clauses used to generate this learned clauses (through resolution). Deletion according to a particular group ID removes all clauses (both original and learned) having this ID.

For the encoding based approach, we utilize the incremental SAT feature of zChaff and group all CNF clauses associated with the comparator using the same group ID. This implementation enable us to only change a very small fraction of all clauses (both original and learned) that are related to the comparator for each different value of k during binary search or linear scan. All clauses associated with the adder are independent of k and hence remain unchanged throughout the entire incremental SAT. Recall that the adder corresponds to many more clauses than the comparator does.

However, unlike the encoding based approach, the diagnosis based approach requires us to update the original clauses by inserting some relaxation variables. This makes it harder to use the incremental SAT algorithm. However, we can still group all the hard constraint clauses in a group and reuse all learned clauses that are generated within this group. In other words, we only delete the learned clauses associated with the relaxable constraint clauses.

4 Experimental Results

We implemented the *subSat* approach for PM-SAT using the chain counter and hierarchical tree counter proposed by Xu [20] for comparison. In addition, we translate our PM-SAT benchmarks into MinCostSat instances so that we can have an extensive comparison using other general purpose solvers. Recall that a MinCostSat problem is a SAT problem with a cost function for each satisfying assignment. We add a unique relaxation variable to each relaxable clause in PM-SAT and the cost of this relaxation variable is 1. All other variables have a cost of 0. Non-relaxable clauses remain unchanged in the above translation. The resulting problem is now a MinCostSat instance where the minimum cost corresponds to the minimum number of relaxation variables setting to be 1, which in turn implies that minimum number of clauses are relaxed. We then use *Scherzo* [3], *bsolo* [13] and *cplex* [7] to solve the translated MinCostSat problem. *Scherzo* is a well known branch-and-bound solver for Binate/Uniate Covering Problem (BCP/UCP) that incorporates many state-of-the-art techniques, including Maximum Independent Set [5] based lower bounding, branch variable selection and various search pruning rules. The BCP problem is essentially a MinCostSat problem [12] with a specific cost function. UCP has the additional restriction that all variables appear in only one phase. But unfortunately, *Scherzo* is not able to solve any of the benchmarks in the following tables. *bsolo* is another state-of-the-art branch-and-bound BCP/UCP solver based on the SAT solver GRASP [14]. *cplex* is the cutting edge commercial Linear Programming (LP) solver that is also capable of finding integer solutions efficiently.

All the experiments are conducted on a Dell PowerEdge 700 running Linux Fedora core 1.0 (g++ GCC 3.3.2) with single Pentium 4 2.8GHz, 1MB L2 cache CPU on 800MHz main bus.

4.1 FPGA Routing Benchmarks

We conduct our experiments mainly on industrial benchmarks. Table 1 shows the results of industrial examples resulting from a SAT based FPGA router. Each relaxable clause corresponds to a net-arc (single source, single destination) in the routing problem. Relaxation of clauses in the unsatisfiable SAT instance to make it satisfiable represent finding the fewest number of net-arcs which, if re-routed elsewhere, e.g. route-around, would allow the remaining set of net-arcs to be routed simultaneously.

The benchmark name in Table 1 shows the number of net-arcs in the actual FPGA routing problem. For example, the first row shows the result of benchmark `FPGA_27`, which has 27 net-arcs to be routed. The PM-SAT instance consists of 3953 Boolean variables with 13537 clauses. Among these clauses, 27 clauses are marked as relaxable,

Table 1. Performance comparison on FPGA routing benchmarks. Timeout for all solvers: 1 hour. * indicates server times out, the best solution found is reported.

Bench-mark	Num. Vars.	Num. Cls.	Rlx. Cls.	Min. k	Diagnosis	Encoding		subSat		Gen. Solver	
					Core Rmv	Binary	Linear	Chain	Tree	bsolo	cplex
FPGA_27	3953	13537	27	3	1.85	2.13	1.65	2.64	2.85	21.29	3*
FPGA_31	17869	65869	31	1	380.83	88.68	309.75	393.26	860.18	4*	12*
FPGA_32	2926	9202	32	3	0.89	1.10	0.95	1.12	1.18	6.56	3*
FPGA_33	9077	32168	33	3	18.65	19.25	26.44	27.02	27.93	61.5	4*
FPGA_39	6352	22865	39	4	31.22	7.76	7.15	8.83	8.48	59.07	6*
FPGA_44	6566	22302	44	3	10.12	9.00	8.36	11.75	12.80	6*	5*

which corresponds to the 27 net-arcs to be routed. The optimal solution is a relaxation of $k = 3$ clauses (out of 27) that makes the entire problem satisfiable. The diagnosis based approach takes 1.85 seconds to find the optimal solution. The binary search and linear scan of the encoding based approach take 2.13 and 1.65 seconds respectively. The *subSat* approach using linear scan with the chain counter and hierarchical tree counter need 2.64 and 2.85 seconds respectively. For the translated MinCostSat problem, *bsolo* takes 21.29 seconds. *cplex* only reports a solution of $k = 3$ but it cannot prove its optimality. *Scherzo* could not report any solution found within the 1 hour time limit for all our benchmarks and hence is omitted from all of our tables.

4.2 Multiple Property Checking Benchmarks

Table 2 shows the results of multiple property checking using circuits from `ISCAS85` and `ITC99` benchmarks. Relaxable clauses are the properties (assertions) that assume each output signal of the entire circuit to be 1 or 0. The non-relaxable clauses are translated from the circuit structure. The corresponding PM-SAT instance is to find the maximum number of outputs that can be 1 or 0 (satisfying the property). Benchmarks that are satisfiable without any relaxation are excluded from the tables. All benchmarks start with a `c` are from the `iscas85` family and the rests are from the `itc99` family.

4.3 Randomized UNSAT Benchmarks

Table 3 shows the results of classic UNSAT benchmarks with randomly chosen relaxable clauses. These benchmarks are from the `fvp-unsat-2.0` (verification of superscalar microprocessors) family by Velev [19]. Note that all the benchmarks in Table 3 have $k = 1$, which makes it inefficient to use binary search.

Table 1, Table 2 and Table 3 clearly show that both approaches constantly outperform the best known solvers. For benchmarks with a large number of relaxable clauses, e.g. `b17`, `b20` and `b22` in Table 2 and all benchmarks in Table 3, the diagnosis based approach has obvious advantage over the search approach (either binary or linear), which suffers from the large auxiliary adder. With most other benchmarks like `c6288` in Table 2 and `FPGA_39` in Table 1, whose number of relaxable clauses is small, the encoding based search approach is faster. As we can see from the tables that there is no significant difference between binary search and linear scan used in the encoding based

Table 2. Performance comparison on multiple property checking benchmarks. Benchmarks end with _1 (_0) are asserted to be 1 (0). Timeout for all solvers: 1 hour. * indicates server times out, the best solution found is reported.

Bench-mark	Num. Vars.	Num. Cls.	Rlx. Cls.	Min. k	Diagnosis Core Rmv	Encoding Binary	Encoding Linear	subSat Chain	subSat Tree	Gen. Solver bsolo	Gen. Solver cplex
c2670_1	1426	3409	140	7	0.05	0.07	0.06	0.25	0.34	1.05	106.40
c5315_1	2485	6816	123	10	0.09	0.18	0.13	0.83	1.13	15.35	208.80
c6288_1	4690	11700	32	2	343.71	81.98	192.27	185.67	169.94	2*	3*
c7552_1	4246	10814	108	5	2.64	1.41	1.17	1.62	1.77	5*	1909.63
b14_1	10044	28929	245	1	0.19	0.53	0.48	0.79	0.93	4.45	1182.85
b15_1	8852	26060	449	2	0.26	1.08	1.07	1.23	1.85	7.55	1308.47
b17_1	32229	94007	1445	6	1.65	14.93	14.92	67.73	90.85	65.88	65*
b20_1	20204	58407	512	2	0.50	2.62	2.90	1.97	5.59	10.88	5*
b21_1	20549	59532	512	2	0.49	2.59	2.89	2.36	5.47	12.63	1522.35
b22_1	29929	86680	757	4	0.96	6.19	5.43	11.78	18.52	25.53	31*
c7552_0	4246	10814	108	6	1.57	2.54	1.99	2.07	1.95	2369.5	6*
b15_0	8852	26060	449	3	0.22	0.14	0.79	1.83	2.68	19.37	260.75
b17_0	32229	94007	1445	13	4.54	13.74	6.85	90.64	173.17	17*	13*

Table 3. Performance comparison on randomized UNSAT benchmarks. The encoding based approach using binary search is inefficient since $k = 1$ for all benchmarks and hence omitted. Timeout for all solvers: 1 hour. * indicates server times out, the best solution found is reported.

Bench-mark	Num. Vars.	Num. Cls.	Rlx. Cls.	Min. k	Diagnosis Core Rmv	Encoding Linear	subSat Chain	subSat Tree	Gen. Solver bsolo	Gen. Solver cplex
2pipe	892	6695	6695	1	5.12	22.34	34.82	65.54	868.34	1*
3pipe	2468	27533	5470	1	4.96	18.18	19.97	31.43	1*	1*
4pipe	5237	80213	802	1	8.45	8.65	8.81	11.32	1*	1*
5pipe	9471	195452	19474	1	18.91	305.69	273.79	367.93	1*	1*
6pipe	15800	394739	15828	1	107.33	383.36	463.81	424.35	1*	1*

approach. However, we still believe that for instances with large k, binary search is a better option. In addition, for the benchmarks with small k value, the performance of the *subSat* approach is comparable with our encoding based approach. This is because the $\lceil \lg k + 1 \rceil$ bit adder used in *subSat* is not much larger than a full adder or half adder for very small k, e.g. $k = 2$. However, the performance *subSat* of degrades dramatically with relative large k, e.g. benchmark `b17_0` in Table 2.

It is interesting to see that all SAT based approaches (all of our approaches, *subSat* and *bsolo*) generally outperform the non-SAT based branch-and-bound methods like *scherzo* and *cplex*. One possible reason is that these industrial benchmarks have more implications and conflicts, than the typical UCP/BCP or ILP instances.

Both our diagnosis based and encoding based approaches benefi t from the incremental SAT and so does our implementation of the *subSat*. Among these three, the encoding based approach gains the most improvements due to the incremental SAT as it solves very *similar* SAT instances. However, we could not provide additional results for this due to the page limit.

Note that though both our diagnosis based and encoding based methods use zChaff as an underlying SAT solver, each of them has a customized zChaff solver based on the features needed. The zChaff solver used in the encoding based approach is relatively faster than the one used in the diagnosis based approach as the latter one has a significant overhead of bookkeeping the information for constructing an UNSAT core. Further, different approaches solve different numbers of underlying SAT instances. The binary search always makes $\lceil \lg n+1 \rceil$ SAT calls while the core elimination approach and the linear search make $k+1$ such SAT calls. The overall time used as presented in all tables includes the time used by these intermediate SAT instances. Usually the SAT instance becomes more and more difficult as we approach k, which is because the corresponding instance becomes more and more constrained.

Usually the diagnosis based approach has an advantage for instances with large n *and small* k *due to the absence of the overhead caused by the hierarchical adder. For instances with relatively small* n*, the encoding based approach is faster and particularly binary/linear search should be used when* k *is large/small.*

5 Conclusions and Future Directions

We have presented two complete and efficient approaches specialized for solving the PM-SAT problem, which arises from various situations including multiple property checking, FPGA routing, etc. These two specialized solvers significantly outperform the best known solvers.

Some key features about these two approaches can be summarized as follows:

1. The diagnosis based approach uses an iterative core elimination technique, which does not require any auxiliary structure and hence is independent of the total number of relaxable clauses. This approaches iteratively identifies the UNSAT core of the problem and relaxes it by inserting relaxation variables to the relaxable clauses in the core. We provide a proof of optimality for this approach.
2. The encoding based approach uses an auxiliary counter implemented as an efficient hierarchical tree adder with a logic comparator to constrain the number of true relaxation variables, and hence the number of relaxed clauses, during the search. The hierarchical tree adder only needs to be constructed once at the beginning and the SAT solver can re-use most of the learned clauses for the instances generated during each search iteration using incremental SAT techniques. Both binary and linear search are supported in this approach.

We believe that there is still room for improvement. One such area is the further tuning of zChaff solver according to different characteristics of the SAT instances generated by each of the two approaches. The other area is to optimize the UNSAT core generation process, e.g. reducing overhead, minimizing UNSAT core, etc.

Acknowledgement

We would like to thank Richard Rudell, Olivier Coudert and Vasco Manquinho for providing us various solvers and benchmarks.

References

1. J. Argelich and F. Manyà. Solving over-constrained problems with SAT technology. *Lecture Notes in Computer Science (LNCS): Theory and Applications of Satisfi ability Testing: 8th International Conference*, 3569:1–15, 2005.
2. B. Cha, K. Iwama, Y. Kambayashi, and S. Miyazaki. Local search algorithms for partial MAXSAT. In *Proceedings of the Fourteenth National Conference on Artifi cial Intelligence*, pages 263–268, 1997.
3. O. Coudert. On solving covering problems. In *Proceedings of the 33rd Design Automation Conference*, pages 197–202, 1996.
4. N. Eén and N. Sörensson. MiniSat – A SAT solver with conflict-clause minimization. In *Proceedings of the International Symposium on the Theory and Applications of Satisfi ability Testing*, 2005.
5. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freemand & Co., New York, 1979.
6. E. Goldberg and Y. Novikov. Berkmin: A fast and robust SAT solver. In *Proceedings of the Design Automation and Test in Europe*, pages 142–149, 2002.
7. ILOG. Cplex homepage, `http://www.ilog.com/products/cplex/`, 2006.
8. D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.
9. R. H. Katz. *Contemporary Logic Design*. Benjamin Cummings/Addison Wesley Publishing Company, 1993.
10. H. Kautz, B. Selman, and Y. Jiang. A general stochastic approach to solving problems with hard and soft constraints,, In D. Du, J. Gu, and P. M. Pardalos, editors, The Satisfiability Problem: Theory and Applications, 1996.
11. I. Koren. *Computer Arithmetic Algorithms*. Prentice Hall, 1993.
12. X. Y. Li. *Optimization Algorithms for the Minimum-Cost Satisfi ability Problem*. PhD thesis, Department of Computer Science, North Carolina State University, Raleigh, North Carolina, 2004. 162 pages.
13. V. Manquinho and J. Marques-Silva. Search pruning techniques in SAT-based branch-and-bound algorithms for the binate covering problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21:505–516, 2002.
14. J. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.
15. S. Miyazaki, K. Iwama, and Y. Kambayashi. Database queries as combinatorial optimization problems. In *Proceedings of the International Symposium on Cooperative Database Systems for Advanced Applications*, pages 448–454, 1996.
16. 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*, 2001.
17. L. Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2004.
18. O. Strichman. Prunning techniques for the SAT-based bounded model checking problem. In *Proceedings of the 11th Conference on Correct Hardware Design and Verifi cation Methods*, 2001.
19. M. Velev. Sat benchmarks, `http://www.ece.cmu.edu/~mvelev/`, 2006.
20. H. Xu. *subSAT: A Formulation for Relaxed Satisfi ability and its Applications*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2004. 160 pages.
21. L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of the Design Automation and Test in Europe*, 2003.