A Fast Pseudo-Boolean Constraint Solver

Donald Chai[1]   Andreas Kuehlmann[1,2]

[1] University of California at Berkeley, CA, USA
[2] Cadence Berkeley Labs, Berkeley, CA, USA

Abstract

Linear Pseudo-Boolean (LPB) constraints denote inequalities between arithmetic sums of weighted Boolean functions and provide a significant extension of the modeling power of purely propositional constraints. They can be used to compactly describe many discrete EDA problems with constraints on linearly combined, parameterized weights, yet also offer efficient search strategies for proving or disproving whether a satisfying solution exists. Furthermore, corresponding decision procedures can easily be extended for minimizing or maximizing an LPB objective function, thus providing a core optimization method for many problems in logic and physical synthesis. In this paper we review how recent advances in satisfiability (SAT) search can be extended for pseudo-Boolean constraints and describe a new LPB solver that is based on generalized constraint propagation and conflict-based learning.

Categories and Subject Descriptors:
I.2.3 [Deduction and Theorem Proving]: Inference engines;
G.1.6 [Optimization]: Integer programming
General Terms: Algorithms, Verification
Keywords: satisfiability, pseudo-Boolean, 0-1 ILP

1 Introduction

Recent advances in solving Boolean satisfiability problems caused a significant resurgence of their application in multiple EDA domains. For example, bounded model checking is based on solving a series of SAT formulas which represent finite unfolding of the design to be checked. Only the latest improvements in SAT search [1, 2] made this method practical in comparison with early approaches, e.g. in sequential test pattern generation.

The efficiency of modern SAT solvers can be attributed to three key features: (1) fast Boolean Constraint Propagation (BCP) based on effective filtering of irrelevant parts of the problem structure, (2) learning of compact facts representing large infeasible parts of the solution space, and (3) fast selection of decision variables. All three features heavily exploit the simple structure of a SAT problem represented in conjunctive normal form (CNF) – a conjunction of multiple constraints each being a disjunction of literals.

Although SAT is generally useful for propositional decision problems, other families of constraints such as *linear pseudo-Boolean (LPB) constraints* can encode many EDA problems more compactly. LPB constraints have the form $\sum a_i \cdot l_i \geq k$; $a_i, k \in \mathbb{R}$; l_i is the variable x_i or its negation $\bar{x}_i$ and $x_i \in \{0, 1\}$. The special case

$\sum l_i \geq k$ is also denoted as *cardinality constraints*. Binate covering-based technology mapping, constraint-based placement and routing [3, 4], current estimation and timing and noise analysis can directly apply LPB constraints to encode feasible solutions. Furthermore, an additional pseudo-Boolean objective function of the form $\sum a_i l_i \rightarrow$ max can be maximized by a series of decision problems making a corresponding solver applicable for optimization problems from the above mentioned domains.

Conventionally, LPB problems are handled by generic Integer Linear Program (ILP) solvers. The drawback is that they typically ignore the Boolean nature of the variables and thus cannot apply specialized methods for efficient constraint propagation and pruning of the search space. On the other hand, LPB decision problems could be encoded as pure CNF-SAT instances which are then solved by any of the highly specialized SAT approaches. However, the number of clauses required for expressing the LPB constraints is large [5], and moreover, a pure CNF encoding may prevent the solver from effectively processing the search space. For example, the pigeonhole problem states that $n + 1$ pigeons cannot be placed in n holes without sharing. The length of the shortest resolution proof of unsatisfiability of the corresponding CNF problem is exponential in the number of holes [6]. Therefore, every Davis-Putnam-Logeman-Loveland-style (DPLL) solver [7, 8] will exercise an exponential runtime. In contrast, a description based on cardinality constraints suits this problem naturally and the length of the shortest cutting plane proof [9, 10] of unsatisfiability is only quadratic [11].

All modern, general purpose SAT solvers are based on the DPLL [12, 8] backtrack search procedure and apply conflict-based learning to derive new clauses for representing an abstraction of unsatisfiable parts of the solution space. This learning mechanism effectively implements a heuristic for scheduling individual resolution steps to assist the backtrack search. In this paper we describe how this scheduling scheme can be generalized for cutting plane proofs and thus be applied for problems that include LPB constraints. We further present a generalized watch-literal strategy [2] which is applicable for Boolean constraint propagation on LPB constraints.

2 Preliminaries
2.1 Constraints

A 0-1 ILP constraint is an inequality of the form:

$$\sum_i a_i \cdot x_i \geq b, \ \ a_i, b \in \mathbb{R}, \ x_i \in \{0, 1\} \tag{1}$$

A constraint is satisfied under some assignment of values to the variables if the respective inequality holds. Using the relation $\bar{x}_i = (1 - x_i)$, the general form of (1) can be converted into an equivalent normalized LPB constraint with only positive coefficients:

$$\sum a_i \cdot l_i \geq k, \ \ a_i, k \in \mathbb{R}^+, \ l_i \in \{x_i, \bar{x}_i\}$$

While in theory, constraints may have real-valued coefficients, we make the same assumption as in [5], that constraints are integer-valued. This simplifies the implementation but does not prevent its application to real-valued problems which can be encoded by integer coefficients in a straightforward manner.

The right-hand side k of a normalized LPB constraint is called its *degree*. An LPB constraint in which all $|a_i|$ are equal is also known as a cardinality constraint, since it merely requires that the number of true literals be greater than or equal to some k':

$$\sum l_i \ge \left\lceil \frac{k}{a_0} \right\rceil = k'$$

A cardinality constraint with $k = 1$ is equivalent to a conventional CNF clause, i.e.,

$$\sum l_i \ge 1 \iff \bigvee l_i$$

Formally speaking, an LPB constraint is a hyperplane in Boolean space. As demonstrated, a LBP inequality is more expressive than a CNF clause, but less expressive than a unate function, since the function $(x_1 \wedge x_2) \vee x_3 \vee x_4$ can be expressed by a single LBP constraint, but not $(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$.

2.2 Operations on Constraints

A proof for demonstrating satisfiability or unsatisfiability of a set of constraints is based on a sequence of inference steps using operations on individual constraints. The primary inference step for CNF clauses is *resolution* [12] which combines a pair of clauses in which exactly one literal l appears positively in one clause and negatively in the other, i.e.,

$$\frac{\begin{array}{c} l_1 \vee \cdots \vee l_k \vee l \\ l_1' \vee \cdots \vee l_m' \vee \bar{l} \end{array}}{l_1 \vee \cdots \vee l_k \ \vee \ l_1' \vee \cdots \vee l_m'}$$

Here we adopted the notation that the antecedents are shown above the line and the consequences below the line.

The operation on LPB constraints which corresponds to CNF clause resolution is *cutting planes* [9, 10] and computes a non-negative linear combination of a set of LPB constraints, optionally rounding coefficients up afterward. For example, combining two constraints in non-normalized notation (i.e., form (1)) yields:

$$\frac{\begin{array}{c} \lambda \cdot (\sum a_i \cdot x_i \ge b) \\ \lambda' \cdot (\sum a_i' \cdot x_i \ge b') \end{array}}{\lambda \cdot \sum a_i \cdot x_i + \lambda' \cdot \sum a_i' \cdot x_i \ge \lambda \cdot b + \lambda' \cdot b'}$$

As an example, the application of $\lambda = 1$ and $\lambda' = 2$ in conjunction with $\bar{x}_3 = 1 - x_3$ eliminates x_3 in the following:

$$\frac{\begin{array}{c} 1(x_4 + 3x_5 + 2x_3 \ge 3) \\ 2(x_1 + x_2 + \bar{x}_3 \ge 2) \end{array}}{2x_1 + 2x_2 + x_4 + 3x_5 \ge 5}$$

The coefficients of an LPB constraint may be rounded up, i.e.,

$$\frac{\sum a_i \cdot x_i \ge b}{\sum \lceil a_i \rceil \cdot x_i \ge \lceil b \rceil}$$

Correctness of rounding follows from $\lceil a \rceil + \lceil b \rceil \ge \lceil a + b \rceil$. For example, by multiplying the constraint with $\lambda = 1/3$ the following rounding can be performed:

$$\frac{\begin{array}{c} 3x_1 + x_2 + x_3 + x_4 + x_5 \ge 6 \\ x_1 + \frac{1}{3}x_2 + \frac{1}{3}x_3 + \frac{1}{3}x_4 + \frac{1}{3}x_5 \ge 2 \end{array}}{x_1 + x_2 + x_3 + x_4 + x_5 \ge 2}$$

Saturation is a corollary of rounding, i.e., all coefficients a_i saturate at k. This can be shown by repeatedly multiplying a constraint with some λ between $\frac{k-1}{k}$ and 1 and rounding until all $a_i \le k$. For example:

$$\frac{\begin{array}{c} 0.6(3x_1 + x_2 + x_3 \ge 2) \\ 1.8x_1 + 0.6x_2 + 0.6x_3 \ge 1.2 \end{array}}{2x_1 + x_2 + x_3 \ge 2}$$

Reduction [13] on an LPB constraint reduces coefficients on the left-hand side and reduces the degree accordingly. For example, reduction can be used to remove x_4 and x_{28} in the following:

$$\frac{x_1 + x_2 + x_4 + 2x_7 + 2x_{28} \ge 5}{x_1 + x_2 + 2x_7 \ge 2}$$

Cardinality constraint reduction derives a cardinality constraint from a general LPB constraint [13]. This is done by successively accumulating the sum of the by magnitude sorted set of coefficients starting from the largest a_i, to detect the minimum number of terms that must be satisfied for fulfilling the constraint. For example:

$$\frac{6x_1 + 5x_2 + 4x_3 + 3x_4 + 2x_5 + x_6 \ge 17}{x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \ge 4}$$

This is because a successive check of $\sum_{i=1}^{n} a_i < 17$ holds true for $n = 1, 2, 3$ but not for $n = 4$. In addition, some of the literals with the smallest a_i may be safely removed from the derived cardinality clause. This is illustrated more clearly by first performing a reduction step to eliminate x_6 from the original constraint followed by a regular cardinality constraint reduction:

$$\frac{\begin{array}{c} 6x_1 + 5x_2 + 4x_3 + 3x_4 + 2x_5 + x_6 \ge 17 \\ 6x_1 + 5x_2 + 4x_3 + 3x_4 + 2x_5 \ge 16 \end{array}}{x_1 + x_2 + x_3 + x_4 + x_5 \ge 4}$$

The detailed procedure for cardinality clause reduction is given in Section 3.2.3.

3 Proof Procedures

A general procedure for proving unsatisfiability of a SAT instance applies a sequence of resolution steps until an empty clause is derived [12]. This procedure is complete but in general requires a number of steps that is exponential in the number of variables. Even in the practically common case that a polynomial-length proof exists, finding the actual schedule of the individual resolution steps is difficult. The most effective SAT solvers apply a DPLL-style backtracking procedure, which systematically searches the solution space by making successive assignments to variables.

For a given partial variable assignments, Boolean constraint propagation (BCP) generates a set of implied assignments which must hold for the SAT instance to be satisfied under the current partial assignment. If these assignments do not have any value conflicts, the search continues until a complete assignment is found and thus satisfiability demonstrated. If, however, a conflict occurs, conflict analysis – also denoted as "learning" – computes a new clause which represents an abstraction of the conflict context. This is done by selectively applying resolution to a set of clauses of interest.

The general DPLL algorithm with learning as implemented by Chaff [2] is shown in Algorithm 1. The backtrack search procedure is organized as a depth-first traversal through the decision tree, where each node is a value assignment for a particular decision variable. The decision level of an assignment is the length of the path from the root to that assignment.

Algorithm 1 DPLLSATSEARCH

---
while (MAKEDECISION() $\neq$ *DONE*)
  while (BCP() = *CONFLICT*)   // find stable assignment
    if (CONFLICTANALYSIS() = *CONFLICT*)
      return *UNSAT*
return *SAT*

There are two subtleties in the implementation of Algorithm 1 that dramatically increase the performance of SAT solvers. First, the routine BCP must efficiently filter the clauses to be processed during Boolean constraint propagation. This is crucial since the SAT search spends the majority of the processing time in this part. Second, the outlined procedure does not explicitly "flip" decision variables. This is accomplished by constructing a conflict clause during CONFLICTANALYSIS that includes exactly one literal on the current decision level. The following BCP step will then automatically drive the SAT search into the yet unexplored part.

General solvers for ILP problems are based on successive applications of cutting plane steps in a series of non-integral relaxations. The idea to solve LPB problems efficiently is to combine the DPLL-style decision procedure given in Algorithm 1 with the application of cutting plain steps for driving the search direction. The main contribution of this paper is to generalize the two above mentioned key elements of efficient solver implementations for LPB constraints. The following two sections outline an approach for fast BCP for LPB constraints and a cutting plane algorithm for conflict analysis that ensures continuous progress during the search.

3.1 Boolean Constraint Propagation

In SAT, the *unit clause rule* ensures whenever one literal in a clause is unassigned and all others evaluate to false, BCP deduces that this literal must be assigned to true. The fastest known method for BCP in SAT is based on the *watch-literal* strategy [2] which avoids a significant fraction of unnecessary clause processing but still guarantees that all unit clauses are identified and corresponding implications are processed. This strategy exploits the fact that a clause cannot trigger an implication as long as two of its literals remain unassigned. It is implemented by processing or "watching" just two arbitrarily chosen literals per clause. Whenever, during the SAT search, either watch-literal is assigned to false it is swapped with a different unassigned literal. If no such literal exists an implication is generated for the second unassigned watch-literal.

A generalized version of BCP for LPB constraints is based on the idea that a literal is implied as soon as its coefficient must be included for satisfying the constraint. For a formal analysis, let s denote the *slack* of a constraint such that:

$$s = \sum_{l_i \neq \text{false}} a_i \; - \; k \tag{2}$$

where k denotes the degree of the constraint. Informally, the slack is the maximal amount by which the constraint can be over-satisfied assuming that all unassigned literals are true. Then for any unassigned literal l_i such that

$$s - a_i < 0 \tag{3}$$

l_i is implied under the current variable assignment. For the following example, the assignment $\bar{x}_1$ results in a slack $s = 3$ which implies x_2 and $\bar{x}_3$:

$$\frac{6x_1 + 5x_2 + 4\bar{x}_3 + 2x_4 + x_5 \geq 9 \;\wedge\; x_1 = 0}{x_2 = 1 \;\wedge\; x_3 = 0}$$

In the following we describe how the watch-literal strategy can be extended for LPB constraints. The idea of literal watching is to minimize the monitoring effort for a constraint while still being able to detect the precise moment when a constraint is violated or some of its unassigned literals are implied.

To ensure that a constraint is not violated, it is sufficient to watch a set of true or unassigned literals L_w such that

$$\sum_{i \in L_w} a_i = S_w \geq k$$

where S_w denotes the watch sum. This is because all the literals of L_w could still be asserted and thus satisfy the constraint.

Detecting implied assignments requires one step of lookahead. To ensure that no variable assignments may be implied, it is sufficient to watch a set of true or unassigned literals L_w such that

$$\sum_{i \in L_w} a_i = S_w \geq k + a_{\max} \tag{4}$$

where $a_{\max}$ denotes the largest coefficient of any unassigned literal (or anything larger). This ensures that there is no unassigned literal l_i such that $a_i > s \geq (S_w - k) \geq a_{\max}$.

Whenever a literal of L_w is assigned to false, it will be removed from L_w and new, true or unassigned literals are added to L_w until condition (4) holds. If this cannot be accomplished, the literals $l_{max} \in L_w$ are successively implied until (4) finally holds.

Algorithm 2 BCP triggered by watched literal $l_t = \text{false}$

L_w: Set of watched literals of constraint
L_u: Set of non-watched, non-false literals of constraint
S_W: Watch Sum

$L_w \leftarrow L_w \setminus \{l_t\}$
$S_w \leftarrow S_w - a_t$
$a_{max} \leftarrow \max\{a_i \mid l_i \in L_w \cup L_u \;\wedge\; l_i \neq \text{true}\}$
while ($S_w < k + a_{max} \;\wedge\; L_u \neq \emptyset$) // fill watch set
  $a_s \leftarrow \max\{a_i \mid l_i \in L_u\}$
  $S_w \leftarrow S_w + a_s$
  $L_w \leftarrow L_w \cup \{l_s\}$
  $L_u \leftarrow L_u \setminus \{l_s\}$
if ($S_w < k$)   // detect conflict
  return *CONFLICT*
while ($S_w < k + a_{max}$)   // detect implications
  IMPLY(l_{max})
  $a_{max} \leftarrow \max\{a_i \mid l_i \in L_w \;\wedge\; l_i \neq \text{true}\}$
return *NO_CONFLICT*

Algorithm 2 shows the procedure for BCP processing of an LPB constraint that was triggered by assigning a watched literal to false. The following example illustrates this procedure for the constraint:

$$6x_1 + 5x_2 + 5\bar{x}_3 + 3x_4 + 2x_5 + 2x_6 + x_7 \geq 12$$

Suppose initially $L_w = \{x_1, x_2, \bar{x}_3, x_4\}$, thus $S_w = 19 \geq 12 + 6$. Next the assignment $x_3 = 1$ triggers a swapping of watched literals, resulting in $L_w = \{x_1, x_2, x_4, x_5, x_6\}$ and $S_w = 18 \geq 12 + 6$. The next assumed assignment $x_4 = 0$ will first swap the remaining literals into the watch set such that $L_w = \{x_1, x_2, x_5, x_6, x_7\}$. The resulting condition $S_w = 16 < 12 + 6$ causes then the implication $x_1 = 1$ after which $S_w = 16 < 12 + 5$. This again implies $x_2 = 1$ which finally satisfies $S_w = 16 \geq 12 + 2$. Note that at this point the constraint is not satisfied yet. The remaining watch set $S_w = \{x_1, x_2, x_5, x_6, x_7\}$ is used for a continued monitoring of the unassigned literals.

There are several adjustments of Algorithm 2 for overall efficiency. For example, the re-filling of the watch set can be done in any order as long as the literal l_{max} is included in L_w. Furthermore, a conservative watch threshold $k + a_{max}$ can be used by selecting the largest constraint coefficient, independent of the assignments.

3.2 Conflict-based Learning

Learning was proposed in [14] as a method to record "no-goods", i.e., partial assignments that are guaranteed to cause a conflict, and further extended in [1]. For a CNF-based SAT solver, a "learned" clause can be deduced by a sequence of resolution steps which combine the clauses leading to the conflict in reverse order of the original implication sequence. To aid this analysis, an implication graph is used which records the actual assignment causalities and the timing during BCP. The sequence of clauses generated by the resolution steps corresponds to a sequence of cuts in the implication graph each separating the "conflict side" from the causing assignments. Under the current assignment, any of these clauses can reproduce the conflict and thus could be added to the CNF formula as learned "no-good".

GRASP [1] introduced a particularly important detail about the cut selection which significantly contributes to the efficiency of modern SAT solvers. Instead of choosing a clause corresponding to an arbitrary cut, GRASP chooses a clause that includes exactly one literal on the current decision level; all remaining literals are from higher levels. This selection guarantees, that the value of the variable on the current decision level is implied after backtracking, thus avoiding an explicit "flipping" of decision variables. This scheme leads to the compact flow outlined in Algorithm 1 where MAKEDECISION asserts an unassigned variable to either 1 or 0; the complemented space is then automatically searched through learning a conflict clause and BCP.

In the following we present how a learning scheme that drives the search forward can be adopted for pseudo-Boolean constraints. As outlined before, the operation on linear constraints that corresponds to CNF clause resolution is cutting planes which computes a linear combination of a pair of LPB constraints to eliminate a particular variable. The problem with general cutting plane operations is that they might weaken the conflict situation and thus not necessarily imply a forward leading variable assignment.

We use the following set of LPB constraints to illustrate the general problem and to outline the suggested solution:

$$
\begin{array}{lll}
(a) & 3x_1 + 2x_2 + x_7 + 2\bar{x}_8 & \geq 3 \\
(b) & 3\bar{x}_1 + x_3 + x_5 + x_9 & \geq 3 \\
(c) & \bar{x}_2 + \bar{x}_3 + x_6 & \geq 2
\end{array} \qquad (5)
$$

Suppose that x_8 was assigned to 1 on some previous decision level and x_6 was set to 0 on the current level. Figure 1 shows the implication graph for BCP processing for this example. The "@" notation indicates the timing of the individual implication steps.
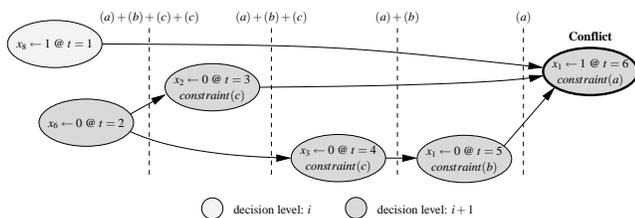

Figure 1: Implication graph for conflict-based learning example.

The application of cutting plane steps (using $\bar{x} = 1 - x$) eliminates the variables x_1, x_3, and x_2 in reverse order of their implications. This results in the following sequence of constraints which correspond to the cuts indicated in Figure 1.

$$
\begin{array}{llll}
t = 4 : (a) + (b) & 2x_2 + x_7 + 2\bar{x}_8 + x_3 + x_5 + x_9 & \geq 3 \\
t = 3 : \quad +(c) & x_2 + x_7 + 2\bar{x}_8 + x_5 + x_9 + x_6 & \geq 3 & (6) \\
t = 2 : \quad +(c) & \bar{x}_3 + x_7 + 2\bar{x}_8 + x_5 + x_9 + 2x_6 & \geq 4
\end{array}
$$

Note that the original pair of constraints (a) and (b) is in conflict for the given partial assignment $(\bar{x}_2, \bar{x}_3, \bar{x}_6, x_8)$, however, the result of the first cutting plane step combining the two at $t = 4$ is not. This is because a single cutting plane step is a weakening operation and may not preserve the full information available in the original set of constraints [15]. This is particularly problematic if the resulting learned constraint is too weak to "flip" the variable on the current decision level which is essential for driving the search forward in the outlined flow. In the given example, the partial assignment $x_8 \wedge \bar{x}_6$ necessarily leads to the conflict, thus the fact $\bar{x}_8 \vee x_6$ could be asserted. However, this information is not preserved in the last constraint learned at $t = 2$, which represents the cutting plane for the given partial assignment, i.e., x_8 does not imply x_6.

The slack as defined in equation 3 can be used to analyze the cause of the constraint weakening and to control the cutting plane steps such that the resulting constraint remains strong enough to drive the search forward. The following gives the slack of the individual constraints in reverse order of the implication process:

$$
\begin{array}{cccc}
& \text{Constraint} & \text{Partial assignment} & \text{Slack} \\
t = 5 & 3x_1 + 2x_2 + x_7 + 2\bar{x}_8 \geq 3 & \bar{x}_1, \bar{x}_2, \bar{x}_3, \bar{x}_6, x_8 & -2 \\
t = 4 & 3\bar{x}_1 + x_3 + x_5 + x_9 \geq 3 & \bar{x}_2, \bar{x}_3, \bar{x}_6, x_8 & +2 \\
t = 3 & \bar{x}_2 + \bar{x}_3 + x_6 \geq 2 & \bar{x}_2, \bar{x}_6, x_8 & \pm 0 \\
t = 2 & \bar{x}_2 + \bar{x}_3 + x_6 \geq 2 & \bar{x}_6, x_8 & \pm 0
\end{array}
$$
$$(7)$$

Clearly, the combined slack will be 0 after processing the first two constraints, which is the reason why the resulting constraint is not in conflict anymore for the partial assignments at $t = 4, 3, 2$.

In general, the constraint at the conflict will always have a negative slack. During the cutting plane steps, only the addition of a positive slack constraint may weaken the result such that the conflict information is lost. This can be avoided by reducing all constraints to be added beforehand such that the combined slack will remain negative. For example, constraint (b) can be reduced by first removing x_9 followed by saturating x_1 resulting in:

$$
(b') \quad 2\bar{x}_1 + x_3 + x_5 \geq 2
$$

which has a slack of $+1$. The resulting cutting plane sequence is:

$$
\begin{array}{llll}
t = 4 : 2(a) + 3(b') & 4x_2 + 2x_7 + 4\bar{x}_8 + 3x_3 + 3x_5 & \geq 6 \\
t = 3 : \quad +3(c) & x_2 + 2x_7 + 4\bar{x}_8 + 3x_5 + 3x_6 & \geq 6 & (8) \\
t = 2 : \quad +(c) & \bar{x}_3 + 2x_7 + 4\bar{x}_8 + 3x_5 + 4x_6 & \geq 7
\end{array}
$$

Note that the resulting learned constraint at $t = 2$ is strictly stronger than $\bar{x}_8 \vee x_6$ and will "flip" x_6 as desired.

Key to the outlined approach is to reduce each constraint to be added by removing unassigned variables followed by saturation such that the combined slack after the cutting plane step remains negative. This process is guaranteed to work since a repeated reduction of constraints will eventually lead to a simple CNF clause with a slack of 0. However, as shown in the example, a complete CNF reduction can often be avoided leading to stronger constraints.

Algorithm 3 CONFLICTANALYSIS for conflicting constraint c_{conf}

I: Set of implications preceding c_{conf} in conflict graph;
     sorted in reverse BCP processing order
c_i: Constraint causing implication i
x_i: Variable asserted by implication i
d_i: Decision level of implication i
A_d: Set of assignments made on decision levels $1 \dots d$

$c \leftarrow c_{conf}$
while ($I \neq \emptyset$)
   $i \leftarrow$ REMOVENEXT (I)
   $c \leftarrow$ REDUCE1 (c)
   $c' \leftarrow$ REDUCE2 (c_i)
   $c \leftarrow$ CUTRESOLVE (c, c', x_i)
   if ($A_{d_i - 1}$ triggers literal implication in c)
     $c \leftarrow$ REDUCE3 (c)
     LEARN c
     BACKTRACK to smallest d such that A_d implies literal in c
     return *NO_CONFLICT*
return *CONFLICT*

Algorithm 3 gives the formal high-level procedure for computing a learned constraint. Starting from the conflicting constraint, the procedure processes the implication graph in reverse topological order by applying repeated cutting plane steps (CUTRESOLVE). It stops when the assignments of the previous decision levels imply at least one literal of the learned constraint. The operation RE-DUCE transforms a constraint via the rules outlined in Section 2.2 such that the combined constraint remains in conflict. The following sections discuss three options for learning constraints based on variants of the REDUCE operations. In Section 5 we present a comparison of the different methods based on detailed benchmarking.

3.2.1 Learning CNF Clauses

In this method, the REDUCE1 and REDUCE2 operations perform a plain CNF reduction on a constraint by combining the implied literal and the literals of the constraint causing the implication into a single CNF clause. If there is a choice in literals, we exploit idempotency for obtaining shorter clauses by choosing literals already known to be in our "conflict clause". This operation is safe and will by construction preserve the property that the resolved clauses remain in conflict. This approach fits smoothly into a CNF based SAT solver as proposed in [5], however, the significant reduction generally causes a redundant processing of search space and a corresponding performance reduction as evaluated in Section 5.

3.2.2 Learning LPB Constraints

This most general form of learning attempts to preserve a maximum strength is the learned constraint by applying careful reduction steps. Key for the REDUCE2 operation is to analyze the set of unassigned literals of the two constraints to be combined and maximally exploit cancellation of complementary literals, which are double counted in slack computations. As shown in the example the actual reduction is then performed by successively removing true or unassigned literals followed by saturation until the combined constraint is expected to retain negative slack. Further, for LPB learning, operations REDUCE1 and REDUCE2 also deal with possible coefficient overflows by pro-actively reducing them.

3.2.3 Learning Cardinality Constraints

Cardinality constraints provide a middle ground between CNF clauses and general LPB constraints with arbitrary coeffi-

cients [13]. In comparison with the latter, cardinality constraints offer a compact representation and fast processing of BCP and conflict-based learning because of their uniform structure. The corresponding learning scheme first generates a LBP constraint as outlined in the previous section and then reduces the result to a cardinality constraint in REDUCE3. The formal pseudo-code for cardinality reduction is presented in Algorithm 4. As illustrated by the example in Section 2.2, the algorithm first collects the minimum number of literals needed for satisfying the original constraint. It then drops as many of the low-coefficient literals as possible.

Algorithm 4 CARDINALITYREDUCTION of constraint c

L: Set of literals in constraint c

$s \leftarrow 0$
$k' \leftarrow 0$
$L' \leftarrow L$
while ($s < k \ \wedge \ L \neq \emptyset$)   // collect minimum number of l_i
   $a_{max} \leftarrow \max\{a_i \mid l_i \in L\}$
   $L \leftarrow L \setminus \{l_{max}\}$
   $s \leftarrow s + a_{max}$
   $k' \leftarrow k' + 1$
$slack \leftarrow \min\{k, \min\{a_i \mid l_i \in L'\} - 1\}$
while ($\min\{a_i \mid l_i \in L'\} - 1 < slack$) // drop l_i with smaller a_i
   $a_{sel} \leftarrow$ SELECT($\{a_i \mid l_i \in L', \ a_i < slack\}$)
   $slack \leftarrow slack - a_{sel}$
   $L' \leftarrow L' \setminus \{l_{sel}\}$
return $\sum_{l_i \in L'} l_i \geq k'$ // constraint with $a_i' = 1$

Since this algorithm performs a weakening, it may happen that a "learned" conflict constraint is no longer in conflict, by changing the relative importance of literals. We conservatively detect this beforehand by checking if $s + \sum_{l_i = \mathsf{false}}(a_i - 1) \geq 0$. If this holds, we reduce away all true and unassigned literals before performing a cardinality reduction.

4 Previous Work

Barth [13] was one of the first to investigate the application of modern DPLL-style search procedures for solving special cases of ILP instances and introduced the concept of cardinality reduction for fast constraint processing. In [5] a pseudo-Boolean solver based on Chaff is described which uses counters to detect logical implications and conflicts. The presented technique does not apply a watch literal scheme for BCP of LPB constraints and uses only simple learning of plain CNF clauses. The authors of [16] integrated a plain cutting plane method into a DPLL-style constraint solver. However, if the resulting constraint is too weak, their approach reverts to full CNF clause reduction.

This paper extends existing work in two critical areas. First, we present a generalized watch-literal scheme for fast BCP of LPB constraints. This technique is similarly critical for a high performance LPB solver as the two-watch-literal strategy for Chaff [2]. Second, we give a general algorithm for LPB learning based on cutting plane operations which guarantees forward progress without the explicit need to generate CNF clauses and avoids unnecessary weakening of the learned facts.

5 Experimental Results

We have implemented a new LPB solver, Galena, in C++ which incorporates all the features described earlier and random restarts [2]. Clause deletion [2] is not implemented. An initial

Benchmark	Vars	Clauses/	Runtime (s)			
		Cardinality	CNF	CARD	LPB	OSL
grout-4.3-1	672	1872/77	0.11	0.21	0.07	1.89
grout-4.3-2	648	1803/77	15.25	0.1	0.65	2.09
grout-4.3-3	648	1796/80	4.96	0.32	0.79	4.31
grout-4.3-4	696	1940/80	4.39	0.11	0.1	1.25
grout-4.3-5	720	1992/88	0.06	0.04	0.04	8.79
grout-4.3-6	624	1750/69	0.46	0.13	0.06	14.74
grout-4.3-7	672	1875/76	0.16	0.08	0.4	1.27
grout-4.3-8	432	1176/71	0.2	0.03	0.05	2.27
grout-4.3-9	840	2330/96	76.63	0.11	0.29	1.73
grout-4.3-10	840	2343/90	0.25	0.13	0.07	0.45
acc-tight:0	1620	1548/468	0.04	0.03	0.05	15.87
acc-tight:1	1620	1827/738	0.18	0.24	0.6	524.84
acc-tight:2	1620	1944/855	98.63	0.21	0.34	*
acc-tight:3	1620	2673/855	0.12	1.69	0.73	*
acc-tight:4	1620	2691/891	16.66	0.67	0.23	*
acc-tight:5	1335	2319/1010	5.98	6.05	24.8	*
acc-tight:6	1335	2321/1003	1.05	3.44	127.2	*

* Timeout (1200s)

Table 1: Decision problem runtimes for various learning options.

set of experiments showed that in our implementation a watching scheme is beneficial for clauses and cardinality constraints, but not for LPB constraints; therefore we use counters to implement Boolean constraint propagation on LPB constraints. All experiments are performed on a Pentium-III 800MHz with 256MB of RAM and 256KB of L2 cache running Linux 2.4.18.

The benchmarks we run include the more difficult global routing benchmarks from [5] and scheduling benchmarks from [17]. Table 1 gives the results for solving these decision problems. The routing benchmarks were run through a trivial preprocessor to obtain a more compact form [16]. The three main columns "CNF", "CARD", and "LPB" correspond to the different learning schemes presented in Sections 3.2.1, 3.2.3, and 3.2.2, respectively. For each method, the initial constraints consist of clauses and cardinality constraints. The column headings for the runtimes refer only to the type of *learned* constraints; the column denoted by "OSL" reports the results obtained from IBM's OSLv3 [18] ILP solver.

As the results in Table 1 show, the LPB and CARD learning schemes generally perform better than CNF, since they may exploit some of the structure in a problem. Both also dramatically outperform OSL in finding a solution to a constraint satisfaction problem. For nontrivial problems, the CARD scheme outperforms the LPB scheme because of the overhead of manipulating LPB constraints.

To increase the difficulty of the benchmarks, we run the same routing examples, with the objective to minimize the number of 1's in the solution. This is done in our solver by incrementally solving a set of decision problems in a linear search. The results are given in Table 2. Here, it is clear that CARD is the overall best learning scheme of the ones presented. Under the CNF scheme, the solver frequently runs out of memory, because it learns many

Benchmark	Optimum	Runtime (s)			
		CNF	CARD	LPB	OSL
grout-4.3-1	62	+	3.5	869	0.88
grout-4.3-2	64	1071	24.19	39.25	1.55
grout-4.3-3	62	*	5.94	739	0.84
grout-4.3-4	60	+	8.05	519	0.85
grout-4.3-5	60	+	17.25	*	0.32
grout-4.3-6	66	88.57	9.06	90.1	1.85
grout-4.3-7	64	44.61	6.09	7.17	1.03
grout-4.3-8	36	*	3.48	234	0.43
grout-4.3-9	68	238	3.73	4.42	1.32
grout-4.3-10	70	4.34	0.76	1.21	1.15

* Timeout (1200s), + Out of memory

Table 2: Optimization runtimes for various learning options.

weak clauses. For optimization, the OSL solver runs faster than all other methods, and outperforms itself when compared to the "easier" decision problems. The former may be due to the high symmetry in the benchmarks, and the latter to OSL's reliance on an objective to drive the search.

6 Conclusions

In this paper we presented a fast pseudo-Boolean constraint solver which is based on generalizing multiple concepts learned from modern SAT solvers. In particular, we described how efficient Boolean constraint propagation using the watch-literal strategy can be extended for pseudo-Boolean constraints and how the general DPLL search scheme can be adopted to drive a cutting plane proof for this class of problems. Our experimental results show that the presented constraint solver outperforms existing approaches and thus may offer an attractive new approach to solve many EDA problems that can be modeled with a combination of classical CNF clauses and pseudo-Boolean constraints.

Our LPB optimizer is still slow in comparison with a commercial ILP solver. In our future work we plan to improve this by using a binary search on the objective and tuning the decision heuristic using an LP solver.

7 Acknowledgments

We would like to thank Fadi Aloul for providing the routing benchmarks, the anonymous referees for their valuable comments, and Robert Brayton for useful discussions.

References

[1] J. P. Marques-Silva and K. A. Sakallah, "GRASP: a search algorithm for propositional satisfiability," *IEEE Trans. on Computers*, vol. 48, no. 5, pp. 506–521, 1999.

[2] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of the 38th ACM/IEEE Design Automation Conference*, (Las Vegas, Nevada), pp. 530–535, June 2001.

[3] S. Devadas, "Optimal layout via Boolean satisfiability," in *IEEE/ACM Inter'l Conference on CAD*, pp. 294–297, November 1989.

[4] R. G. Wood and R. A. Rutenbar, "FPGA routing and routability estimation via Boolean satisfiability," *IEEE Trans. on VLSI Systems*, pp. 222–231, June 1998.

[5] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, "Generic ILP versus specialized 0-1 ILP: An update," in *IEEE/ACM Int. Conference on Computer-Aided Design*, pp. 450–457, November 2002.

[6] A. Haken, "The intractability of resolution," *Theoretical Computer Science*, vol. 39, pp. 297–308, 1985.

[7] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of of the Assoc. of for Computing Machinery*, vol. 7, pp. 102–215, 1960.

[8] M. Davis, G. Logeman, and D. Loveland, "A machine program for theorem proving," *Communications of the ACM*, vol. 5, pp. 394–397, July 1962.

[9] R. Gomory, *An algorithm for integer solutions to linear programs.* New York: McGraw-Hill, 1963.

[10] V. Chvátal, "Edmonds polytopes and a hierarchy of combinatorial problems," *Discrete Mathematics*, vol. 4, pp. 305–337, 1973.

[11] H. E. Dixon and M. L. Ginsberg, "Combining satisfiability techniques from AI and OR," *The Knowledge Engineering Review*, vol. 15, pp. 31–45, March 2000.

[12] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of the ACM*, vol. 7, pp. 102–215, 1960.

[13] P. Barth, *Logic-Based 0-1 Constraint Programming.* Kluwer Academic Publishers, 1995.

[14] R. M. Stallman and G. J. Sussman, "Forward reasoning and dependency directed backtracking in a system for computer aided circuit analysis," *Artificial Intelligence*, vol. 9, no. 2, pp. 135–196, 1977.

[15] V. Chandru and J. Hooker, *Optimization Methods for Logical Inference.* New York: John Willey & Sons, Inc., 1999.

[16] H. E. Dixon and M. L. Ginsberg, "Inference methods for a pseudo-Boolean satisfiability solver," in *National Conference on Artificial Intelligence*, 2002.

[17] J. P. Walser, "0-1 integer optimization benchmarks." http://www.ps.uni-sb.de/ walser/benchmarks/benchmarks.html.

[18] M. S. Hung, W. O. Rom, and A. D. Waren, *Optimization with IBM OSL.* Scientific Press, 1993.