Efficient SAT engines for concise logics: Accelerating proof search for zero-one linear constraint systems

Martin Fränzle[1] * and Christian Herde[2] **

[1] Informatics and Mathematical Modelling, The Technical University of Denmark
Richard Petersens Plads, Bldg. 322, DK-2800 Kgs. Lyngby, Denmark
[2] Department of Computer Science, Carl-von-Ossietzky Universität Oldenburg
P.O. Box 2503, D-26111 Oldenburg, Germany

Abstract. We investigate the problem of generalizing acceleration techniques as found in recent satisfiability engines for conjunctive normal forms (CNFs) to linear constraint systems over the Booleans. The rationale behind this research is that rewriting the propositional formulae occurring in e.g. bounded model checking (BMC) [5] to CNF requires a blowup in either the formula size (worst-case exponential) or in the number of propositional variables (linear, thus yielding a worst-case exponential blow-up of the search space). We demonstrate that acceleration techniques like observation lists and lazy clause evaluation [14] as well as the more traditional non-chronological backtracking and learning techniques generalize smoothly to Davis-Putnam-like resolution procedures for the very concise propositional logic of linear constraint systems over the Booleans. Despite the more expressive input language, the performance of our prototype implementation comes surprisingly close to that of state-of-the-art CNF-SAT engines like ZChaff [14]. First experiments with bounded model-construction problems show that the overhead in the satisfiability engine that can be attributed to the richer input language is often amortized by the conciseness gained in the propositional encoding of the BMC problem.

Keywords: Satisfiability; non-clausal propositional logic; zero-one linear constraint systems; proof search; acceleration techniques

1 Introduction

As many verification and design automation problems concerning finite state hardware and software systems can be expressed as satisfiability problems for propositional logics, there is a growing demand for efficient satisfiability checkers for such logics. Application domains include combinational and sequential

* Email: `mf@imm.dtu.dk`,
Phone/Fax: +45-4525 7512/+45-4593 0074.
** Email: `Christian.Herde@Informatik.Uni-Oldenburg.De`,
Phone/Fax: +49-441-798-2160/+49-441-798-2145.

equivalence checking for circuits, test pattern generation, FPGA routing, and bounded model-checking, i.e. the verification of a temporal property on a finite unraveling of a transition system [5]. As these are enabling technologies for tackling the complexity of ever larger circuits and ever more refined embedded software, and as furthermore even the size of the propositional formulae derived from current devices is prohibitive, this has sparked much research on enhancing satisfiability checkers and on generating more compact propositional encodings of the problem statements.

Concerning the performance of propositional satisfiability (SAT) checkers, the most dramatic improvements have recently been accomplished on SAT solvers for conjunctive normal forms (CNFs) that implement variants of the Davis-Putnam-Loveland-Logemann (DPLL) procedure [7]. As in the classical DPLL procedure, the main algorithmic ingredients of these solvers are unit propagation and backtrack search. These have, however, been enhanced by heuristics for finding a suitable traversal sequence through the backtrack-search tree, as well as by refined algorithms and data structures for pruning the search tree and for accelerating unit propagation. Considerable search-tree pruning has been achieved through non-chronological backtracking [9, 13, 20, 14] and conflict-driven learning [13, 20, 14], sometimes combined with random restart [3, 14]. Unit propagation is sped up through dedicated data structures [21, 20] and through lazy clause evaluation [14], which delays re-evaluation of the truth value of any clause that is definitely non-unit.

While these techniques actually yield a dramatic speedup in practice, now tackling instances with hundreds of thousands of propositions, which would have been completely impractical a decade ago, they still reach their limits for state-of-the-art verification problems derived from high-level design models (e.g., Statemate models [10]) of embedded software. Such models easily yield CNFs with millions of propositional variables under bounded model checking, even if BDD-based Boolean-function minimization is applied to the transition relation in a preprocessing step [11]. While some of this complexity is inherent to the verification task, another part can be attributed to the low expressiveness of conjunctive normal forms. Rewriting arbitrary propositional formulae to CNF yields a worst-case exponential blowup in formula size if the number of propositional variables is to be preserved. To avoid this, all practical verification environments take advantage of satisfiability-preserving transformations that yield linear-size encodings through introduction of a linear number of auxiliary variables [17, 15, 18]. The price for introducing a linear number of auxiliary variables is, however, a worst-case exponential blow-up in the size of the backtrack-search tree.

Yet, it has been observed that both causes of blow-up can often be avoided, as the DPLL procedure generalizes smoothly to zero-one linear constraint systems (the constraint parts of zero-one linear programs) [4, 19, 1]. Zero-one linear constraint systems are expressive enough to facilitate a linear-size encoding of, e.g., gate-level netlists without use of auxiliary variables. Following the seminal work of Barth, who generalized the basic DPLL procedure [4], Whittemore, Kim, and Sakallah tried to follow up the advances in the algorithmics of CNF SAT solvers by adapting GRASP's conflict analysis and learning to zero-one linear constraint systems [19], and recently Aloul, Ramani, Markov, and Sakallah ported CHAFF's lazy clause evaluation to this setting [1]. Yet, they simply mimicked CHAFF's lazy evaluation scheme such that their type of lazy clause evaluation is confined to the pure CNF part of the problem, i.e. applies only to those clauses that are disjunctive. As all other clauses are evaluated eagerly, and as clause re-evaluation is known to account for the major part of a DPLL-based SAT solver [14], this is far from optimal. In this paper, we will show that it is possible and effective to generalize lazy clause evaluation to arbitrary linear constraints under a zero-one interpretation. A similar method has come to our knowledge after submission of this report: Chai and Kuehlmann have independently designed and implemented closely related techniques [6].

We will provide a brief introduction to zero-one linear constraint systems in Section 2 and to the state of the art in CNF SAT in Section 3. Section 4 explains the algorithms and data structures underlying our SAT solver for zero-one linear constraint systems, which incorporates generalized DPLL, conflict-driven learning, and lazy clause evaluation for general linear constraint clauses. Section 5, finally, provides benchmark results.

2 Zero-one linear constraint systems

Given the fact that CNF encoding of formulae is often substantially more complex — either wrt. formula size or wrt. number of auxiliary propositional variables — than encoding in more general classes of propositional formulae, we take advantage of propositional formulae in conjunctive clause form that have a much more liberal notion of clause. In this class, called *zero-one linear constraint systems (ZOLCS)* or *linear pseudo-Boolean constraint systems* [4], formulae are conjunctions of linear threshold clauses. A *linear threshold clause* is of the form

$$a_1 x_1 + a_2 x_2 + \dots a_n x_n \geq k \ ,$$

where the x_i are *literals*, i.e. positive or negated *propositional variables*, the a_i are natural numbers, called the *weights* of the individual literals, and $k \in \mathbf{N}$ is the *threshold*.

Given a Boolean valuation of the propositional variables, a threshold clause is satisfied iff its left hand side evaluates to a value exceeding the threshold when the truth values `false` and `true` are identified with 0 and 1, respectively. I.e.,

$$5a + 3\overline{b} + 3c + 1d \geq 7 \ ,$$

where $\overline{b}$ denotes the negation of b, is satisfied by the valuation

$$a \mapsto \texttt{false}, b \mapsto \texttt{false}, c \mapsto \texttt{true}, d \mapsto \texttt{true} \ ,$$

yet is not satisfied by the valuation

$$a \mapsto \texttt{true}, b \mapsto \texttt{true}, c \mapsto \texttt{false}, d \mapsto \texttt{true} \ .$$

Threshold clauses can represent a wide class of monotonic Boolean functions, e.g. $1a + 1b + 1\overline{c} + 1d \geq 1$ is equivalent to $a \vee b \vee \overline{c} \vee d$, $1a + 1b + 1\overline{c} + 1d \geq 4$ is equivalent to $a \wedge b \wedge \overline{c} \wedge d$, and $1a + 1b + 3\overline{c} + 1d \geq 3$ is equivalent to $c \implies (a \wedge b \wedge d)$. Consequently, zero-one linear constraint systems can be exponentially more concise than CNF: a CNF expressing that at least n out of k variables should be true requires $\binom{n}{k}$ clauses of length n each, i.e. is of size $O\left(\binom{n}{k}n\right)$, whereas the corresponding ZOLCS has size linear in k and logarithmic in n.

Formally, the syntax of zero-one linear constraint systems is

$$\begin{aligned}
\mathit{formula} &::= \{\mathit{clause} \wedge\}^* \mathit{clause} \\
\mathit{clause} &::= \mathit{linear_term} \geq \mathit{threshold} \\
\mathit{linear_term} &::= \{\mathit{weight}\ \mathit{literal} +\}^* \mathit{weight}\ \mathit{literal} \\
\mathit{weight} &::\in \mathbf{N} \\
\mathit{literal} &::= \mathit{var} \mid \overline{\mathit{var}} \\
\mathit{var} &::\in V \\
\mathit{threshold} &::\in \mathbf{N}
\end{aligned}$$

where V is a countable set of propositional variable names. In accordance with the correspondence of certain linear threshold clauses to standard Boolean connectives, we say that a zero-one linear constraint system ϕ is *in CNF form* iff all the weights and all the thresholds occurring in ϕ are equal to 1.

ZOLCS' are interpreted over Boolean *valuations* $\sigma : V \stackrel{\text{total}}{\longrightarrow} \mathbf{B}$ of the propositional variables. We say that σ *satisfies* ϕ, denoted $\sigma \models \phi$, iff σ satisfies all clauses of ϕ. σ satisfies a clause $a_1x_1 + a_2x_2 + \ldots a_nx_n \geq k$ iff $a_1\chi_\sigma(x_1) + a_2\chi_\sigma(x_2) + \ldots a_n\chi_\sigma(x_n) \geq k$, where

$$\chi_\sigma(x) = \begin{cases} 0 \text{ if } x \in V \text{ and } \sigma(x) = \texttt{false}, \\ 1 \text{ if } x \in V \text{ and } \sigma(x) = \texttt{true}, \\ 1 \text{ if } x \equiv \overline{y} \text{ for some } y \in V \text{ and } \sigma(y) = \texttt{false}, \\ 0 \text{ if } x \equiv \overline{y} \text{ for some } y \in V \text{ and } \sigma(y) = \texttt{true}. \end{cases}$$

When solving ZOLCS satisfiability problems with Davis-Putnam-like procedures, we will build valuations incrementally such that we have to reason about *partial valuations* $\rho : V \stackrel{\text{part.}}{\longrightarrow} \mathbf{B}$ of the propositional variables. We say that a variable $v \in V$ is *unassigned in* ρ iff $v \notin \text{dom}(\rho)$. A partial valuation ρ is called *consistent for a formula* ϕ iff there exists a total extension $\sigma : V \stackrel{\text{total}}{\longrightarrow} \mathbf{B}$ of ρ that satisfies ϕ. Otherwise, we call ρ *inconsistent for* ϕ. Furthermore, a partial valuation ρ is said to *satisfy* ϕ iff all its total extensions satisfy ϕ. As this definition of satisfaction agrees with the previous one on total valuations, we will use the same notation $\rho \models \phi$ for satisfaction by partial and by total valuations.

3 State of the art in clausal SAT

Many modern implementations of complete satisfiability-search procedures are enhancements of the Davis-Putnam-Loveland-Logemann recursive search procedure [7]. Given a CNF ϕ and a partial valuation ρ, which is empty at the start, the DPLL procedure incrementally extends ρ until either $\rho \models \phi$ holds or ρ turns out to be inconsistent for ϕ, in which case another extension is tried through backtracking. Extensions are constructed by either logical deduction based on *unit resolution* or by so-called *decision steps*, which entail selecting an unassigned variable and "blindly" assigning a truth-value to it. The DPLL procedure alternates between these two extension strategies, thereby using unit resolution whenever possible:

DPLL(ϕ, ρ):
  DEDUCE(ϕ, ρ) % Unit resolution
  if ϕ contains conflicting clause **then return**
  if no free variables left **then exit_with**(ρ)
  b := SELECT_UNASSIGNED_VARIABLE(ϕ) % Decision step:
  v := ONE_OF {`true`, `false`}
  DPLL(ϕ, $\rho \cup \{b \mapsto v\}$) % Blind assignment,
  DPLL(ϕ, $\rho \cup \{b \mapsto \neg v\}$) % reversed upon backtrack

Unit resolution, as implemented by DEDUCE, checks for occurrence of so-called *unit clauses* and extends the current partial valuation by their implications. A clause "is unit" iff all its literals, except for exactly one unassigned literal, are set to false. To satisfy the CNF ϕ, the unassigned literal has to be assigned true. That assignment is thus said to be *propagated* by the unit clause. Such propagation is iterated until no further unit clauses exist.

DEDUCE(ϕ, ρ):
  while unit-clause (x) exists in ϕ
    if $x \equiv b$ for some $b \in V$ **then** $\rho := \rho \cup \{b \mapsto \texttt{true}\}$
    if $x \equiv \overline{b}$ for some $b \in V$ **then** $\rho := \rho \cup \{b \mapsto \texttt{false}\}$

Deduction may yield a *conflicting clause* which has all its literals assigned false, indicating a dead end in search. Backtracking then backs up to the most recently assigned decision variable which has not yet been tried both ways, flips its truth-value, and resumes the search. If no such decision variable exists then ϕ is unsatisfiable. If, on the other hand, all variables of ϕ can be assigned without a conflict, a satisfying valuation has been found.

Being based on this recursive search procedure, actual implementations refine the above scheme by using different strategies for selecting the variable and the truth-value to be used at a decision step. See [12] for a detailed review of the impact of different decision strategies on the solving process. Furthermore, state-of-the-art SAT solvers enhance the basic procedure through various algorithmic modifications.

3.1 Conflict analysis and conflict-driven learning

Like all pure backtracking algorithms, the DPLL procedure suffers from thrashing, i.e. repeated failure due to the same reason. To overcome this problem, sufficiently general reasons for the conflict encountered have to be deduced and stored for future guidance of the search procedure. The standard scheme traces the reason back to a small (ideally minimal) number of assignments that triggered the particular conflict, and stores this reason by adding the negation of that assignment as a clause — termed *conflict clause* — to the clause database.

Such reasons can be inferred from cuts in the propagation graph. The *propagation graph* is a directed graph recording the causal relationship between individual variable assignments performed. It is obtained through relating, for each propagation performed, the propagated literal to the previously assigned literals in the corresponding unit clause. A cut in this graph constitutes a reason for all assignments occurring later in the graph. A more general reason, i.e. a reason mentioning fewer literals, for an individual assignment can be derived by taking just those elements of a cut that actually have a path to that particular assignment. Using this technique, reasons for the assignments constituting a conflict can be inferred. The reason is, however, not unique, as different cuts can be used. The interested reader is referred to Zhang et al. [22], who examine several heuristics for deriving one or more conflict clauses from the propagation graph.

3.2 Backjumping

In addition to learning conflict clauses, the result of the conflict diagnosis can also be used to accelerate backtracking: instead of just backtracking to the most recent decision step, the algorithm may immediately undo all decisions up to the latest assignment to a variable occurring in the identified conflict clause, as later decisions do not affect the conflict. Because of this non-sequential way of backing up through the levels of the search tree, this technique is referred to as *backjumping*.

3.3 Restarts

Another enhancement which can be applied in presence of conflict-driven learning is *(random) restarts*. Restarts abort an ongoing search, discard the partial valuation which has been constructed so far, and resume solving from an empty valuation while retaining the learned conflict clauses. This technique is intended to prevent the search from getting lost in non-relevant regions of the search space which might have been entered by early decisions. The restarted solver will not simply repeat its previous run due to the learned conflict clauses, which guide the solver to different parts of the search space. Note, however, that some care has to be taken to preserve completeness of the search process [2], e.g. by limiting the number of restarts.

3.4 Lazy clause evaluation

Concerning performance of DPLL-like procedures, probably the most important improvement in recent years can be attributed to a novel implementation of unit propagation, introduced with the ZChaff sat solver [14]. Unit propagation in general accounts for the major fraction of solver run-time.

Previous implementations of unit propagation identified unit clauses by visiting, after each assignment, *all* clauses containing the literal falsified by that assignment, as such clauses might have become unit. The key idea of the enhanced algorithm is to watch only two literals in each clause, and not to visit the clause when any other literal is assigned as the two watched literals provide evidence of its non-unitness. If an assignment, however, sets a watched literal to false, then this triggers a visit of the respective clause to evaluate its state. The algorithm then tries to replace the watched literal that has been assigned false with another unassigned or true literal occurring in the clause. If it succeeds then it has constructed a new witness for non-unitness of the clause. Otherwise, the clause has become unit and the unassigned one of the watched literals is the one to be propagated.

This technique, often called *lazy clause evaluation*, has been shown to achieve significant performance gains, especially on hard SAT instances [14].

4 Accelerated proof search for ZOLCS satisfiability

We will now generalize DPLL and its recent enhancements, in particular lazy clause evaluation, to zero-one linear constraint systems. To simplify the exposition we assume in the following that each threshold clause is rewritten after each assignment according to the following simplification rules:

- A literal which has been set to false is removed from the lefthand side of the clause as it cannot contribute to its satisfaction any longer. For example, after assigning $b \mapsto \texttt{true}$, the clause $1a + 1\overline{b} + 1\overline{c} \geq 2$ is written as $1a + 1\overline{c} \geq 2$.
- A literal which has become true is also removed from the clause, however with its weight subtracted from the threshold of that clause. For example, after assigning $b \mapsto \texttt{false}$ the clause $1a+1\overline{b}+1\overline{c} \geq 2$ is written as $1a+1\overline{c} \geq 1$.

Consequently, all literals appearing in clauses are yet unassigned, and indices range over unassigned literals only.

4.1 DPLL on threshold clauses

Since the seminal work of Barth [4] it is well-known that the basic DPLL procedure can easily be generalized to zero-one linear constraint systems through modification of the deduction procedure.

As before, the objective of the deduction routine is to detect propagating clauses and to perform the corresponding assignments. A threshold clause $\sum a_i x_i \geq k$ propagates a literal x_j iff setting this literal to false would make the clause unsatisfiable, i.e. iff $(\sum a_i) - a_j < k$.

In contrast to a CNF clause, a general threshold clause can propagate several literals simultaneously. As an example consider the clause

$$5a + 3\overline{b} + 3c + 1d + 1e \geq 7$$

which propagates $\overline{b}$ and c immediately after setting a to false. Carrying out the assignments $b \mapsto \texttt{false}$ and $c \mapsto \texttt{true}$ reduces the clause to

$$1d + 1e \geq 1$$

which shows that, as opposed to CNF-SAT, a threshold clause is not necessarily satisfied after propagation.

We refer to a threshold clause which propagates at least one literal as a *propagating clause*. With this notion, corresponding to the notion of a unit clause in CNF-SAT, we can formulate the generalized DEDUCE procedure for ZOLCS as follows:

DEDUCE(ϕ, ρ):
  while propagating clause $\mathcal{C}$ exists in ϕ
    for each literal x propagated by $\mathcal{C}$
      if $x \equiv b$ for some $b \in V$ **then** $\rho := \rho \cup \{b \mapsto \texttt{true}\}$
      if $x \equiv \overline{b}$ for some $b \in V$ **then** $\rho := \rho \cup \{b \mapsto \texttt{false}\}$

Observe that a propagating clause propagates at least the literal with the largest weight appearing in that clause. If this is not unique due to several literals with the same largest weight, then all those literals are propagated.

This can be demonstrated by the following argument: Let x_p be the (not necessarily unique) literal with the largest weight appearing in a clause $\sum a_i x_i \geq k$. Suppose that some literal x_q with $q \neq p$ is propagated by that clause. Then $(\sum a_i) - a_q < k$ holds due to the propagation rule given above. However, since $c_p \geq c_q$, also $(\sum a_i) - a_p < k$ holds, i.e. x_p is propagated, too.

4.2 Generalization of lazy clause evaluation

While the generalization of the DPLL procedure to ZOLCS has already been proposed by Barth [4], its acceleration through lazy clause evaluation for arbitrary threshold clauses is a novel contribution.[3] To apply lazy clause evaluation to threshold clauses we have to determine a subset of unassigned literals from each clause such that watching these literals is sufficient for detecting change of clause state from normal to propagating. Obviously, we are looking for minimal sets with this property in order to avoid unnecessary clause visits.

To this end, we arrange the literals of each clause with respect to their weights, such that the literal with the largest weight is the leftmost one. Then we read the clause from left to right and select the literals to be watched as follows:

1) The leftmost literal is selected.
2) The following literals are selected until the sum of their weights, not including the weight of the leftmost literal, is greater than or equal to the threshold of the clause.

[3] Aloul et al.'s incorporation of CHAFF's lazy clause evaluation into ZOLCS solving [1] provides lazy clause evaluation for CNF clauses only.

In the example clause below, the literals selected by the aforementioned rules have been marked with grey boxes:

$$\boxed{\boxed{4a} + \boxed{3b} + \boxed{2c} + 1d + 1e + 1f + 1g \quad \geq \ 5}$$

The set of literals chosen to be watched by these rules fulfills the requirements stated above:

a) Assignments which do not affect the watched literals will never make the clause unsatisfiable as the literals selected by rule 2) are by themselves sufficient to ensure satisfiability.
Furthermore, no such assignment will turn the clause into a propagating one. To see this, assume that an assignment to an unwatched literal *does* cause the propagation of some literals. Then, as shown before, the watched literal with the largest weight, selected by rule 1), is among those implications. However, according to the propagation rule from section 4.1, this literal is *not* propagated because the remaining watched literals, selected by rule 2), ensure satisfiability of the clause.
Consequently, a visit of the clause is unnecessary upon assignments affecting unwatched literals only.
b) If, on the other hand, an assignment sets a watched literal to false, the corresponding clause may become a propagating one. Hence, such assignments trigger an evaluation of the clause. However, the set of watched literals guarantees that the clause is satisfiable upon visit as any single literal can be assigned while preserving satisfiability of the clause: if the literal with the largest weight is set to false then the watched literals selected by rule 2) still guarantee satisfiability. If a literal selected according to rule 2) is set to false, the literal selected by rule 1) can compensate for it as its weight is greater than or equal to the one of the literal which has been assigned.
c) The chosen literals form a minimal subset with these properties as we start selecting from the largest weight towards the smallest.

If a watched literal of a clause is assigned false, our algorithm tries to reestablish a set of literals which is in accordance to rules 1) and 2). This requires the search for a minimal set of literals which are either unassigned or true and whose weights sum up to a value that at least equals that of the watched literal which has been assigned false. If such a set exists then it is added to the set of watched literals to replace the one which has dropped out. If no such set exists then this indicates that the clause has become a propagating one. The resulting propagations are determined by application of the propagation rule from section 4.1.

In our implementation, we decided to continue observing those watched literals which have been assigned true as well as those which have been set to false but cannot be replaced by new observations, instead of really removing them from the observation set. Clearly, these observations will not trigger any clause evaluation as long as the corresponding variables keep their values, s.t. there is no need to really remove them. The desired side-effect is, however, that upon *un*assignment of the corresponding variables, these observations are implicitly reactivated. This form of *lazy reactivation* ensures that after backtracking, an appropriate set of literals is watched in each clause. As this is achieved without any computational burden, the time spent on backtracking is linear in the number of unassignments that have to be performed, just as in CNF-SAT. The price to be paid is, however, that after backtracking the observation set of a clause may no longer be minimal, because a single watched literal, which after backtracking is unassigned again, might have previously been replaced by several literals with smaller weights. In principle we could re-establish a minimal set of watched literals by recording the changes in the observation sets triggered by each decision and undoing these changes when backtracking. Yet, this would make backtracking much more expensive and, furthermore, we would abandon the advantage that reassigning a variable shortly after unassigning it – due to the nature of tree-search algorithms a quite frequent case – is usually faster than its previous assignment, as thereafter it is only watched in a small subset of clauses.

Consider again the clause from the previous example. The figure below illustrates the changes of its state and its set of watched literals when successively setting g, c and b to false, then unassigning all variables.

$$\boxed{4a + 3b + 2c + 1d + 1e + 1f + 1g \quad \geq \quad 5}$$

$$\downarrow \; g \mapsto \texttt{false}$$

$$\boxed{4a + 3b + 2c + 1d + 1e + 1f \quad \geq \quad 5}$$

$$\downarrow \; c \mapsto \texttt{false}$$

$$\boxed{4a + 3b + \quad 1d + 1e + 1f \quad \geq \quad 5}$$

$$\downarrow \; b \mapsto \texttt{false}$$

$$\boxed{4a + \quad 1d + 1e + 1f \quad \geq \quad 5}$$

$$\downarrow \; \text{Propagate } a \mapsto \texttt{true}$$

$$\boxed{1d + 1e + 1f \quad \geq \quad 1}$$

$$\downarrow \; \text{Unassign } a, b, c, g$$

$$\boxed{4a + 3b + 2c + 1d + 1e + 1f + 1g \quad \geq \quad 5}$$

4.3 Generalization of learning

Just as in CNF-SAT, conflict-driven learning in our solver for ZOLCS is based on an analysis of the propagation graph as described in section 3.1. For sake of efficiency, the propagation graph is, exactly as e.g. in the ZChaff solver, only maintained implicitly during the search by annotating each variable assigned by propagation with a pointer to its propagating clause, but it is not constructed explicitly until a conflict is really encountered.

To construct the propagation graph, the conflict diagnosis procedure has to determine, for each propagation involved in the conflict, a subset of literals of the corresponding propagating clause whose assignment triggered the propagation. In CNF-SAT, this subset is easy to identify as it consists of *all* literals of the unit clause, except the one which has been propagated. In ZOLCS-SAT, however, we have to take into account that a threshold clause is not necessarily satisfied after propagation and may even become propagating more than once. Hence, the state of its literals at the time of conflict analysis may not coincide its literals' state at the time of propagation. Blindly selecting assigned literals from a clause to reconstruct a cause for a certain propagation may thus introduce acausal relationships or even cycles in the propagation graph.

To avoid this, our implementation associates a timestamp to each variable assignment. The timestamp records the point in time when the variable was assigned; it is reset to zero upon unassignment of the variable. Let $\sum_{i \in M} a_i x_i \geq k$, $M \subseteq \mathbb{N}$ be a threshold clause and x_p a literal propagated by this clause. For each $i \in M$ let t_i denote the timestamp associated with literal x_i. To determine a cause for the propagation of x_p, our implementation chooses a minimal subset of literals with indices $R \subseteq M$ such that $(\sum_{i \in M \setminus R} a_i) - a_p < k$ and $(\sum_{i \in M \setminus R} a_i) + a_p \geq k$ hold and, furthermore, for each $r \in R$ the literal x_r has been assigned false with timestamp $t_r \leq t_p$.

The propagation graph is built by recursively relating propagated literals to their causes, where the latter are determined according to the above rules. Once the propagation graph has been constructed, the algorithm for conflict analysis and learning is equal to the one used in CNF-SAT. In particular, the clauses learned are CNF clauses. We do not support learning of more general clauses right now. Recently, a more general learning scheme learning restricted forms of threshold clauses has been proposed in [6].

5 Benchmark results

In order to evaluate the proposed methods, we have developed Goblin, a new SAT solver for zero-one linear constraint systems, which employs lazy clause evaluation and conflict-driven learning as explained in section 4. For some experiments we used ZChaff, version 2001.2.17, as reference. Goblin, which is written in C++, shares several algorithmic features of the ZChaff solver, however, as opposed to Zchaff, it uses neither random restarts, nor deletion of learned conflict clauses to avoid memory explosion up to now, and it uses a less sophisticated decision strategy. All experiments were performed on a 1 GHz Pentium III machine with 640 MByte physical memory, running Linux.

The first group of experiments dealt with scheduling problems, originally formulated in discrete-time Duration Calculus. The propositional formulae used as input for the SAT engines are bounded model construction problems for Duration Calculus formulae, generated by the method proposed in [8]. Each SAT instance entails the search for a feasible schedule for three periodic tasks, where a feasible schedule corresponds to a satisfying valuation of the respective instance.

The individual instances only differ in the runtime n of the tasks involved in the corresponding task system.

The generation of the input formulae for Goblin exploits that a duration formula of shape $\int y \geq k$, which holds on an observation interval $O = [m, n]$ iff the accumulated duration of y being true over this interval exceeds k, can be directly translated into the threshold clause $\sum_m^n y_i \geq k$, where the y_i represent the values of y in the different time instants.

	CNF			**ZOLCS**		
n	Clauses	Literals	ZChaff [s]	Clauses	Literals	Goblin [s]
1	56758	140637	0.19	21638	109054	0.23
2	107350	273069	0.40	22501	108703	0.25
3	157942	405501	1.97	21467	108193	0.25
4	208534	537933	20.24	21386	107536	0.26
5	259126	670365	153.56	21308	106744	0.25
6	309718	802797	163.45	21233	105829	0.25
7	360310	935229	219.42	21161	104803	0.24
8	410902	1067661	250.35	21092	103678	0.24
9	461494	1200093	277.04	21026	102466	0.24
10	512086	1332525	307.26	20963	101179	0.24

Table 1. Results of scheduling benchmarks.

The results, presented in table 1, show that this encoding yields ZOLCS which have size almost constant in n, whereas the size of the corresponding CNF-formulae grows rapidly with n. The same holds for the solving time. In total, Goblin finished all instances in less than three seconds, whereas ZChaff required more than 23 minutes to complete them.

	Formula	Range	Clauses	Literals
a)	$a^3 + b^3 + c^3 = d^3$	$a, b, c, d \in [100; 200]$	1103	5352
b)	$a^3 + b^3 = c^3$	$a, b, c \in [100; 200]$	827	3952
c)	$a^3 + b^3 = 352 * a * b$	$a, b \in [100; 500]$	1018	4872
d)	$a^3 + b^3 = 517 * a * b$	$a, b \in [100; 500]$	1018	4872
e)	$(416 - a) * b^2 = a^3 \wedge a \neq b$	$a, b \in [100; 500]$	855	3915
f)	$(224 - a) * b^2 = a^3 \wedge a \neq b$	$a, b \in [100; 500]$	855	3915

Table 2. Integer arithmetic problems.

The second group of experiments was carried out to assess the effectiveness of the proposed lazy clause evaluation scheme in terms of the average number of clause evaluations that have to be executed after each assignment performed. To this end, we translated integer arithmetic problems into ZOLCS, using a bit-wise encoding that replaces each occurence of an integer variable a with $\sum_{i=0}^{n-1} 2^i a_i$, where the a_i are propositional variables and n is the number of bits needed to represent a.

Consider for example the integer constraint $a^2 \geq 4$, where a is a 2-bit integer variable. The encoding sketched above yields $(2a_1 + a_0)(2a_1 + a_0) \geq 4$, i.e. $4a_1^2 + 4a_1a_0 + a_0^2 \geq 4$, which can easily be transformed into a ZOLCS by replacing each nonlinear term a_ia_j with an auxiliary variable h and imposing the additional constraint $h \rightarrow a_i \wedge a_j$, where the latter is expressed by $2\overline{h} + a_i + a_j \geq 2$.

Table 2 shows the integer formulae that were used in the experiments, as well as the size of the corresponding ZOLCS.

As a reference for these benchmarks we used a modified version of Goblin that watches *all* literals appearing in a threshold clause, thus mimicking the behaviour of a naive implementation of the DPLL algorithm, that, after each assignment, re-evaluates all clauses containing the literal falsified by that assignment.

Solver Result		Naive Algorithm			Lazy Clause Evaluation		
		Time $[s]$	Assignments Evaluations	EpA	Time $[s]$	Assignments Evaluations	EpA
a)	$a = 108, b = 114,$ $c = 126, d = 168$	94.32	2716535 651362897	239.78	25.88	2853381 23950767	8.39
b)	UNSAT	105.25	2334859 732186876	313.59	21.97	2412304 24734089	10.25
c)	$a = 176, b = 176$	38.35	1273108 212213069	166.69	14.12	1291765 12169638	9.42
d)	UNSAT	268.96	2624583 1268553553	483.36	37.10	2628195 40664770	15.47
e)	$a = 288, b = 432$	8.27	425303 37606297	88.42	3.68	437408 2420138	5.53
f)	UNSAT	19.84	622024 98046695	157.63	6.47	647989 5633296	8.69

Table 3. Results of integer problems.

The results of the experiments are summarized in table 3. "EpA" denotes the average number of clause evaluations per assignment required in the respective solver run. For all benchmarks performed, lazy clause evaluation is able to significantly reduce this value, as well as the runtime of the solver. The largest gain is obtained for benchmark d), where EpA is improved by a factor of 31.2, yielding a speed-up factor of 7.2. The reduction of EpA demonstrates that keeping the observation sets on backtracking does not lead to inefficiently large observation sets. It seems that the lower cost of backtracking gained from this strategy in fact outweighs the clause evaluation overhead caused by non-minimal observation sets, as our scheme provides considerable speedups even on general ZOLCS, like the arithmetic benchmarks. In contrast, Chai and Kuehlmann decided, based on benchmarking their implementation, to constrain lazy clause evaluation to CNF and cardinality clauses (i.e. clauses having only weights of 1) only [6].

6 Discussion

Our first experiments indicate that incorporating state-of-the-art SAT solver algorithmics, in particular lazy clause evaluation, into a ZOLCS satisfiability checker is efficient, with the overhead incurred from solving the more expressive source language being outweighed by the more concise input language. The empiric results are better than expected, given that they are based on a first prototype implementation that is lacking the perpetual refinement of many of its competitors, as well as on a limited set of benchmarks that have not been tuned to fully exploit the conciseness of zero-one linear constraint systems. Especially promising are the results obtained on integer arithmetic problems, a domain traditionally considered to be extremely challenging for propositional solvers, be it BDD-based or SAT-based solvers. It is worth noting that most of the arithmetic operations involved were non-linear such that they are out-of-scope of integer linear programming (ILP) procedures, unless bit-wise encoding is used, which yields poorly performing ILPs in general [16].

Planned optimizations and extensions to our implementation include, besides fine tuning of the various heuristics involved, a more cache-aware implementation which sorts clauses with similar high-weight literals into adjacent memory areas. Furthermore, we plan to exploit the straightforward relaxation of zero-one linear constraint systems into linear programs (LPs) for giving the variable selection strategy used in decision steps a reasonable seed: by occasionally — e.g., upon restarts — solving the relaxation of the ZOLCS with an LP solver, likely truth value assignments for some of the variables can be obtained, which provides guidance for selecting the variable and the initial truth value to be used within decision steps.

Acknowledgment. The authors would like to thank the Safety-Critical Embedded Systems Group of Oldenburg University and the ES/VT division of Kuratorium OFFIS e.V., Oldenburg, for many fruitful discussions.

References

1. Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Generic ILP versus specialized 0-1 ILP: An update. In *Proc. ACM/IEEE Intl. Conf. Comp.-Aided Design (ICCAD)*, pages 450–457, November 2002.
2. L. Baptista, I. Lynce, and J. Marques-Silva. Complete search restart strategies for satisfiability. In *Proc. of the IJCAI'01 Workshop on Stochastic Search Algorithms (IJCAI-SSA)*, August 2001.
3. L. Baptista and J. P. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *Proc. of the 6th International Conference on Principles and Practice of Constraint Programming*, 2000.
4. Peter Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-boolean optimization. Technical Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1995.

5. A. Biere, A. Cimatti, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
6. Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver. In *Proc. of the 40th Design Automation Conference (DAC 2003)*, pages 830–835, Anaheim (California, USA), June 2003. ACM.
7. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.
8. Martin Fränzle. Take it NP-easy: Bounded model construction for duration calculus. In Ernst-Rüdiger Olderog and Werner Damm, editors, *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant systems (FTRTFT 2002)*, volume 2469 of *Lecture Notes in Computer Science*, pages 245–264. Springer-Verlag, 2002.
9. J. F. Groote and J. P. Warners. The propositional formula checker HeerHugo. Technical report SEN-R9905, CWI, 1999.
10. David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts.* McGraw-Hill Inc., 1998.
11. Rainer Lochmann. Boosting the verification power of the statemate verification engine: on the perfomance benefits of integrating ProverCL. In preparation, 2003.
12. J. P. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Proc. of the 9th Portuguese Conference on Artificial Intelligence (EPIA)*, September 1999.
13. 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.
14. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.
15. Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In Alen Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science B.V., 1999.
16. J. H. Owen and S. Mehrotra. On the value of binary expansions for general mixed-integer linear programs. *Operations Research*, 50(5):810–819, 2002.
17. G. Tseitin. On the complexity of derivations in propositional calculus. In A. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logics*, 1968.
18. Joost P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63–69, 1998.
19. Jesse Whittemore, Joonyoung Kim, and Karem Sakallah. SATIRE: A new incremental satisfiability engine. In *Proc. of the 38th Design Automation Conference (DAC 2001)*, pages 542–545, Las Vegas (Nevada, USA), June 2001.
20. H. Zhang. SATO: An efficient propositional prover. In *Proc. of the International Conference on Automated Deduction (CADE)*, volume 1249 of *LNAI*, pages 272–275. Springer, 1997.
21. H. Zhang and M. Stickel. An efficient algorithm for unit-propagation. In *Proc. of the International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96)*, pages 166–169, Fort Lauderdale (Florida USA), 1996.
22. L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proc. of the International Conference on Computer-Aided Design (ICCAD01)*, pages 279–285, November 2001.