Watched Propagation of 0-1 Integer Linear Constraints

Jo Devriendt[1,2](✉)

[1] Lund University, Lund, Sweden
jo.devriendt@cs.lth.se
[2] University of Copenhagen, Copenhagen, Denmark

Abstract. Efficient unit propagation for clausal constraints is a core building block of conflict-driven clause learning (CDCL) Boolean satisfiability (SAT) and lazy clause generation constraint programming (CP) solvers. Conflict-driven pseudo-Boolean (PB) solvers extend the CDCL paradigm from clausal constraints to 0-1 integer linear constraints, also known as (linear) PB constraints. For PB solvers, many different propagation techniques have been proposed, including a counter technique which watches all literals of a PB constraint. While CDCL solvers have moved away from counter propagation and have converged on a two watched literals scheme, PB solvers often simultaneously implement different propagation algorithms, including the counter one.

The question whether watched propagation for PB constraints is more efficient than counter propagation, is still open. Watched propagation is inherently more complex for PB constraints than for clauses, and several sensible variations on the idea exist. We propose a new variant of watched propagation for PB constraints and provide extensive experimental results to verify its effectiveness. These results indicate that our watched propagation algorithm is superior to counter propagation, but when paired with specialized propagation algorithms for clauses and cardinality constraints, the difference is fairly small.

1 Introduction

Although the Boolean satisfiability (SAT) problem is NP-complete [7,19] these days so-called *conflict-driven clause learning (CDCL)* solvers [20,23] routinely solve problems with up to millions of variables. Independently, a similar technique was developed for constraint programming (CP) solvers [2]. These solvers *learn* a propositional disjunction (a *clause*) from each failing search branch, over time accumulating huge databases of clauses that further constrain the search. For example, if, during search, all but one literals of a clause are set to false, the last remaining literal should be *propagated* to true.

To efficiently detect which clauses in the database propagate a literal, modern SAT solvers settled on the *watched literal propagation* technique [23]. Its core idea is to only *watch* two literals of a clause at a time, replacing these *watches* when one or both are set to false. If no two non-falsified watches can be found, the clause either propagates a literal, or it is falsified, indicating a search *conflict*.

© Springer Nature Switzerland AG 2020
H. Simonis (Ed.): CP 2020, LNCS 12333, pp. 160–176, 2020.
https://doi.org/10.1007/978-3-030-58475-7_10

The conflict-driven paradigm has been transferred to *linear pseudo-Boolean (PB) solving*, where solvers deal with linear inequalities over 0-1 integer variables, or *PB constraints* for short.[1] Formulas of PB constraints are a straightforward generalization of the conjunctive normal form (CNF) used for SAT solvers. Crucially, such *conflict-driven PB solvers* learn PB constraints instead of clauses [6,14,18,28], which allows them to construct *cutting planes proofs* [8] instead of the exponentialy weaker *resolution proofs* [4,9,10,25] underlying CDCL.

As the database of learned PB constraints grows during search, and as conflict-driven PB search endeavors to make the learned constraints as strong as possible, the propagation routine forms the main computational bottleneck of PB solvers. Similar to CDCL SAT solvers, a hypothetical doubling the efficiency of PB propagation could translate to almost halving the total run time for conflict-driven PB solvers. Unlike CDCL SAT solvers however, conflict-driven PB solvers have not settled on a dominant propagation scheme.

The *Galena* solver investigated a highly involved watched literal scheme for PB constraints, but finally settled on a three-tiered approach where clauses and *cardinality* constraints were handled with specialized watched propagation techniques, but propagation of general PB constraints was done by *counter* propagation, watching all literals at once [6]. The *Pueblo* solver initially employed the same three-tiered approach [27], but later opted for a custom watched literal scheme [28]. The *Sat4J* system also uses the three-tiered approach by default, but has the option to use watched propagation for general PB constraints similar to the *Galena* watched literal scheme [18]. Finally, the *RoundingSat* solver employs watched propagation, sharing similarities with both the *Pueblo* and original *Galena* approach, but adding its own twists [14].

Unsurprisingly, efficient watched PB propagation is still an open question:

> PB solvers get slower when dealing with pseudo-Boolean constraints because we have not yet found an efficient lazy data structure similar to [...] watched literals for those constraints. This is especially the case for the cutting-planes-based solver because the number of pseudo-Boolean constraints grows during the search [18].

In this paper, we propose a novel efficient watched PB propagation algorithm, and contribute extensive experimental data to shed light on key issues. The general conclusion is that watched PB propagation is more efficient than counter propagation on its own, but that the difference between a counter-based and a watched-based three-tiered approach is fairly small.

This paper continues with preliminaries in Sect. 2 followed by a description of our proposed watched PB propagation algorithm in Sect. 3. Section 4 highlights differences and similarities with the approaches used by the above PB solvers. Experimental results are presented in Sect. 5 and the paper concludes with Sect. 6.

[1] In general, PB constraints can be non-linear, but we restrict our attention to linear PB constraints.

2 Preliminaries

Throughout this paper, we use the term *pseudo-Boolean (PB) constraint* to refer to a 0–1 linear inequality. We identify 1 with *true* and 0 with *false*. A *literal* ℓ denotes either a variable x or its negation $\overline{x}$, where $\overline{x} = 1-x$. We assume without loss of generality that all constraints $\sum_i c_i\ell_i \geq w$ are written in *normalized form*, where literals ℓ_i are over pairwise distinct variables, coefficients c_i are non-negative integers, and w is a positive integer called the *degree (of falsity)*. For a constraint C, $\mathit{lits}(C)$ denotes its set of literals, $\mathit{size}(C)$ its number of literals, and $\mathit{maxcf}(C)$ its largest coefficient. A PB constraint C where $\mathit{maxcf}(C) = 1$ is a *cardinality* constraint and a constraint with degree 1 is a *clause*.

A *(partial) assignment* ρ is a set of literals over pairwise distinct variables. A literal ℓ is *assigned to true* by an assignment ρ if $\ell \in \rho$, *assigned to false* or *falsified* if $\overline{\ell} \in \rho$, and is *unassigned* otherwise. The *slack* of a constraint $C \doteq \sum_i c_i\ell_i \geq w$ under a partial assignment ρ is

$$\mathit{slack}(C,\rho) = -w + \sum_{\overline{\ell}_i \notin \rho} c_i \quad , \tag{1}$$

i.e., the maximal value the left-hand side can attain under any partial assignment $\rho' \supseteq \rho$ minus the degree. We say that ρ *falsifies* C if $\mathit{slack}(C,\rho) < 0$ and *satisfies* C if for any $\rho' \supseteq \rho$ it holds that $\mathit{slack}(C,\rho') \geq 0$. A *pseudo-Boolean formula* φ is a set of PB constraints. An assignment ρ is a *solution* to φ if ρ satisfies all constraints in φ. A formula is *satisfiable* if it has a solution.

A *sequence* $(e_1, \ldots, e_n)$ is a finite ordered collection of elements allowing repetitions.[2] In programming fashion, $\mathit{seq}[i]$ denotes the ith element of seq. The *size* of a sequence is denoted as $\mathit{size}(\mathit{seq})$. A *tuple* is a fixed size sequence with named elements and $\mathit{tup}.e$ refers to the element with name e of tuple tup.[3]

2.1 Conflict-Driven Pseudo-Boolean Solving

We present the bare essentials of conflict-driven PB solving necessary for the discussions in this paper (referring the reader to, e.g., [5] for more details). Conflict-driven PB solving is very similar to the CDCL algorithm for Boolean satisfiability, but uses PB constraints instead of clauses.

The state of a PB solver can be abstractly represented by a tuple (ψ, ρ), where ψ is a set of constraints called the *constraint database* and ρ is a sequence of pairwise distinct literals representing the *current assignment*.[4] Initially, ψ is the input formula φ and ρ is the empty sequence ().

[2] Common data structures for sequences are arrays, lists, and vectors.

[3] Tuples abstract the record data type.

[4] Slightly abusing notation, we defined an assignment as a set, but we often operate on the current assignment ρ as a sequence, *pushing* and *popping* literals from the back.

Given a solver state, the search loop starts with a *propagation* phase, which checks for any constraint $C \in \psi$ whether it is falsified:

$$slack(C, \rho) < 0, \tag{2}$$

or whether a literal ℓ_i, not yet assigned by ρ, in C with coefficient c_i, is implied by C under ρ:

$$slack(C, \rho) < c_i \text{ with } \ell_i \notin \rho, \overline{\ell_i} \notin \rho. \tag{3}$$

If condition (3) holds, C is falsified by $\rho \cup \overline{\ell}_i$, so ℓ_i is implied by C under ρ. Hence, ρ can be safely extended with the implied ℓ_i, which is called a *propagation*, while also saying that C *propagates* ℓ_i. Each propagation can enable new propagations, continuing the propagation phase until condition (3) does not hold for any constraint in the database ψ, or until condition (2) holds for at least one.

If condition (2) holds for some constraint, it is considered a *conflict*, and the solver enters a *conflict analysis* phase. During this phase, the solver derives a *learned constraint* that is a logical consequence of the input formula and would have propagated a literal at some earlier state, preventing the same conflict from happening. This new constraint is added to ψ, after which the solver *backjumps* to the earlier state. Alternatively, if no conflict is detected, the solver extends ρ by making a heuristic *decision* to assign some currently unassigned variable. In either case, the solver continues with a new iteration of the search loop.

The PB solver reports unsatisfiability whenever it learns a constraint equivalent to the trivial inconsistency $0 \geq 1$. If propagation does not lead to a conflict and all variables have been assigned, the solver reports that the input formula is satisfiable. Conflict-driven PB solvers, like their CDCL counterparts, frequently backjump to the root search node, clearing the current assignment from any decision literals and consequent propagations, which is called a *restart*.

In this paper, we focus on the propagation phase, ensuring that after each decision and each backjump, the current assignment is extended with implied literals until fixpoint, or until a conflict arises.

2.2 Counter Pseudo-Boolean Propagation

A straightforward propagation algorithm is the *counter* approach. It takes its inspiration from early SAT propagation algorithms and eagerly computes the slack of each constraint under changes to the current assignment ρ. I.e., each time a literal ℓ is pushed to resp. popped from ρ, due to decisions or propagations resp. backjumps or restarts, each constraint C containing $\overline{\ell}$ has its slack decreased resp. increased with the coefficient of $\overline{\ell}$ in C. When the slack of C is decreased, condition (2) and (3) are checked as well to detect propagations and conflicts.

Example 1. Consider a freshly initialized solver where the input formula consists only of the constraint $C \doteq 3x+2y+z+w \geq 3$. Initially, $\rho = ()$, so $slack(C, \rho) = 4$ and neither condition (2) or (3) hold.

If the solver decides $x = 0$, then $\rho = (\overline{x})$, and counter propagation decreases the slack with 3: $slack(C, \rho) = 1$. Now, condition (3) holds: $slack(C, \rho) < 2$ and $y \notin \rho, \overline{y} \notin \rho$. Hence, y is propagated and $\rho = (\overline{x}, y)$. No further slack decreases are triggered, so counter propagation does not need to check whether condition (2) or (3) hold.

The solver can now decide a new variable, say $z = 0$, so $\rho = (\overline{x}, y, \overline{z})$, at which point counter propagation again decreases the slack of C with 1 to $slack(C, \rho) = 0$, and propagates w, leaving C satisfied by ρ. If the solver instead executes a restart, the current assignment is reset to $\rho = ()$, and counter propagation increases the slack of C with $1 + 3$ to its original value $slack(C, \rho) = 4$.

Unfortunately, counter propagation has the potential for a lot of overhead:

Example 2. Consider a freshly initialized solver where the input formula consists only of the constraint $C \doteq 3x + 2y + z + \sum_{i=1}^{1000} w_i \geq 3$. Let the literals $\overline{w_i}$ be prioritized by the solver's decision heuristic. Initially, $slack(C, \rho) = 1003$, which decreases by 1 after each decision of some w_i. For each of these thousand slack decrements, condition (2) and (3) are never met, since as long as none of x, y and z are falsified by the current assignment, $slack(C, \rho) \geq 3$.

This phenomenon of large amounts of slack decrements (and increments during backjumps) can occur in thousands of constraints simultaneously, considerably slowing down the solver. The watched literal technique attempts to significantly reduce the number of times the slack of a constraint is calculated.

3 Watched Pseudo-Boolean Propagation

Similar observations to those in Example 2 led to the development of *watched (literal)* propagation in SAT solvers [23,30]. This watched approach has been generalized to pseudo-Boolean solving [6,14,18,28]. The central idea of watched PB propagation is to track (*watch*) for each constraint only a subset of its literals – the *watched* literals. The subset is chosen sufficiently large to ensure that as long as none of the watched literals are assigned to false, the constraint is not propagating or conflicting. If one of the watches is assigned false, a search for new non-falsified watches is triggered. If insufficient new watches are found, the constraint may be propagating or conflicting, which is calculated only then.

More formally, we associate each constraint C with a set of watched literals $watches(C)$. For a constraint with watched literals, the *watch slack* of a constraint $C \doteq \sum_i c_i \ell_i \geq w$ under a partial assignment ρ is

$$watchslack(C, \rho) = -w + \sum_{\overline{\ell_i} \notin \rho, \ell_i \in watches(C)} c_i. \tag{4}$$

Clearly, for any C, ρ and watches for C, $watchslack(C, \rho) \leq slack(C, \rho)$, and $watchslack(C, \rho) = slack(C, \rho)$ if all non-watched literals are falsified by ρ. Hence, condition (2) and (3) will never hold (so C will not propagate or be conflicting) if for some set of watches

$$\mathit{watchslack}(C, \rho) \geq \mathit{maxcf}(C). \quad (5)$$

As a result, no exact slack needs to be calculated for constraints for which condition (5) holds, and only by falsifying a watched literal can condition (5) become violated. However, efficiently maintaining appropriate watched literal sets during backjumps, decisions and propagations is a highly non-trivial matter.

To describe our proposed watched PB propagation algorithm in detail, we abstract the state of a constraint C to a tuple $(\mathit{terms}, w, \mathit{wslk})$, where *terms* is a sequence of terms, w a positive integer representing the degree, and *wslk* an integer storing the watch slack of the constraint. The state of a term in *terms* is abstractly represented by a tuple $(\mathit{coef}, \mathit{lit}, \mathit{wflag})$ where *coef* is the coefficient of the term, *lit* the literal, and *wflag* a flag denoting whether the literal is watched for the constraint, i.e., whether $\mathit{lit} \in \mathit{watches}(C)$. We fix *terms* to be sorted in decreasing coefficient order, so $\mathit{maxcf}(C) = C.\mathit{terms}[1]$ – the first term of C contains its largest coefficient.

We also extend the abstraction of the solver state to a tuple $(\psi, \rho, q, \mathit{wlist})$, where the *propagation index* q is an integer s.t. $0 \leq q \leq \mathit{size}(\rho)$[5], and the *watch list wlist* is a function mapping literals to the set of constraints that currently watch the literal combined with the index of the literal in the constraint's term list: $(C, i) \in \mathit{wlist}(\ell)$ iff $\ell \in \mathit{watches}(C)$ with $\ell = C.\mathit{terms}[i].\mathit{lit}$. We define $\rho^i \doteq (\rho[1], \ldots, \rho[i])$ as the *subassignment* up to index i, with $0 \leq i \leq \mathit{size}(\rho)$. The propagation index indicates which part of the current assignment has already been processed for propagation: constraints watching literals in $\rho \setminus \rho^q$ will need to be checked for propagation. Initially, $q = 0$.

3.1 Detailed Algorithm

We now have the necessary abstractions in place to describe our proposed watched PB propagation algorithm in detail. For simplicity, we assume that initially, none of the constraints C are propagating or conflicting, and that their initial watched literals can be chosen to satisfy $\mathit{watchslack}(C, ()) = C.\mathit{wslk} \geq \mathit{maxcf}(C)$.

Procedures `processWatches`, `propagate` and `backjump` present the proposed watched PB propagation algorithm.

Procedure `processWatches` iterates over all literals ℓ in the current assignment, adjusting the watch slack for each constraint C watching $\overline{\ell}$, maintaining the invariant that $\mathit{watchslack}(C, \rho^q) = C.\mathit{wslk}$. It subsequently checks whether C can propagate (or is conflicting) by calling `propagate` for C. If C is conflicting, it is returned. However, breaking out of the loop at line 5 leaves behind a semi-processed set of constraints watching ℓ. To repair this, `processWatches` decreases the propagation index by one, and increases the watch slack for those constraints still watching $\overline{\ell}$ that had their watch slack decreased.

To check whether a constraint is conflicting or propagating, `propagate` first attempts to find non-falsified non-watched literals to use as watches, in the

[5] In *MiniSAT* [13] parliance, q is the *qhead*.

Procedure. propagate(constraint C, integer idx)

```
   External data: watch list $wlist$, current assignment $\rho$
   Result: OK if $C$ is not falsified, otherwise CONFLICT
1  $i \leftarrow 1$
2  while $i \leq size(C)$ and $C.wslk < maxcf(C)$ do
3      $\ell \leftarrow C[i].lit$
4      if $\overline{\ell} \notin \rho$ and $C[i].wflag = 0$ then
5          $C[i].wflag = 1$
6          $wlist(\ell) \leftarrow wlist(\ell) \cup \{(C, i)\}$
7          $C.wslk \leftarrow C.wslk + C[i].coef$
8      $i \leftarrow i + 1$
9  if $C.wslk \geq maxcf(C)$ then
10     $C[idx].wflag = 0$
11     $wlist(C[idx].lit) \leftarrow wlist(C[idx].lit) \setminus \{(C, idx)\}$
12     return OK
13 if $C.wslk < 0$ then return CONFLICT
14 $j \leftarrow 1$
15 while $j \leq size(C)$ and $C.wslk < C[j].coef$ do
16     $\ell \leftarrow C[j].lit$
17     if $\ell \notin \rho$ and $\overline{\ell} \notin \rho$ then $\rho$.push($\ell$)
18     $j \leftarrow j + 1$
19 return OK
```

loop at line 2. If a sufficient amount of watches is found such that $C.wslk \geq maxcf(C)$, no propagation or conflict occurs, the old watch can be discarded at lines 10 and 11, and the routine returns. If all non-falsified literals are employed as watches, yet the watch is still less than zero, the constraint is conflicting, which is returned at line 13. Finally, if the watch slack is non-negative but less than the largest coefficient, the constraint may propagate unassigned literals, which is checked in the loop at line 14. Recall that the terms of a constraint are sorted in decreasing coefficient order, allowing the loop at line 14 to conclude when $C.wslk < C[j].coef$, avoiding a full linear scan. In case $C.wslk < maxcf(C)$, the constraint keeps watching the falsified literal. This allows procedure `backjump` to increase the watch slack of a constraint during backjumps, to a point where $C.wslk \geq maxcf(C)$ without searching for new watches.

3.2 An Extensive Example

Example 3. As in Example 2, consider a freshly initialized solver where the input formula consists only of the constraint $C \doteq 3x + 2y + z + \sum_{i=1}^{1000} w_i \geq 3$. Let the initial watches for C be $\{x, y, z\}$, and hence, $watchslack(C, \rho^q) = C.wslk = 3$. Let the literals $\overline{w_1}$ to $\overline{w_{997}}$ be prioritized by the solver's decision heuristic, so the current assignment ρ is incrementally extended by deciding the literals $\overline{w_1}$ to $\overline{w_{997}}$, and after each decision, procedure `processWatches` is called, incrementing q to 997. As no constraint watches any w_i, `propagate` is never called.

Procedure. processWatches

```
External data: database $\psi$, current assignment $\rho$, propagation index $q$, watch
               list $wlist$
Result: OK if no constraint is falsified, otherwise a falsified constraint
1  while $q < size(\rho)$ do
2      $q \leftarrow q + 1$
3      $\ell \leftarrow \rho[q]$
4      $visited \leftarrow \emptyset$
5      foreach $(C, idx) \in wlist(\overline{\ell})$ do
6          $visited \leftarrow visited \cup \{(C, idx)\}$
7          $C.wslk \leftarrow C.wslk - C[idx].coef$
8          if propagate($C, idx$)$= CONFL$ then
9              $q \leftarrow q - 1$
10             foreach $(C', idx') \in visited \cap wlist(\overline{\ell})$ do
11                 $C'.wslk \leftarrow C'.wslk + C'[idx'].coef$
12             return $C$
13 return $OK$
```

Procedure. backjump(integer s)

```
External data: database $\psi$, assignment $\rho$, propagation index $q$
1  while $size(\rho) > s$ do
2      $\ell \leftarrow \rho[size(\rho)]$
3      if $q = size(\rho)$ then
4          $q \leftarrow q - 1$
5          foreach $(C, idx) \in wlist(\overline{\ell})$ do
6              $C.wslk \leftarrow C.wslk + C[idx].coef$
7      $\rho$.pop()
```

Let $\overline{x}$ be the 998st decision, leading to $\rho = (\overline{w_1}, \ldots, \overline{w_{997}}, \overline{x})$. `processWatches` increases q to 998, and as x is watched by C, decreases $C.wslk$ to 0 and calls `propagate`($C, 1$). `propagate` iterates over C, picking w_{998}, w_{999} and w_{1000} as new watches in the loop at line 2. After exiting the loop, the watch slack has increased to $C.wslk = 3$, so $C.wslk \geq maxcf(C)$ and x is dropped as watch at lines 10 and 11. The watched literals for C now are $watches(C) = \{y, z, w_{998}, w_{999}, w_{1000}\}$.

Let $\overline{z}$ be the 999th literal decision, leading to $\rho = (\overline{w_1}, \ldots, \overline{w_{997}}, \overline{x}, \overline{z})$. Running `processWatches` increases q to 999, and as z is watched by C, decreases $C.wslk$ to 2 and calls `propagate`($C, 3$). `propagate` cannot find further watches, so $0 \leq C.wslk < maxcf(C)$ and the while loop at line 14 looks for literals to propagate. The only literal for which $C.wslk < C[j].coef$, x, is already assigned to false, so no literals can be propagated, and both ρ and $watches(C)$ remain unchanged.

Let $\overline{w_{998}}$ be the next literal decision, leading to $\rho = (\overline{w_1}, \ldots, \overline{w_{997}}, \overline{x}, \overline{z}, \overline{w_{998}})$. `processWatches` increases q to 1000, and as w_{998} is watched, decreases $C.wslk$ to 1 and calls `propagate`($C, 1001$). `propagate` cannot find further watches, so

$0 \leq C.\mathit{wslk} < \mathit{maxcf}(C)$ and the while loop at line 14 checks which literals can be propagated. The only literals for which $C.\mathit{wslk} < C[j].\mathit{coef}$ are x and y. The latter is still unassigned, so it is propagated at line 17. Returning to `processWatches`, $\rho = (\overline{w_1}, \ldots, \overline{w_{997}}, \overline{x}, \overline{z}, \overline{w_{998}}, y)$, so $q = 1000 < \mathit{size}(\rho) = 1001$ – the loop at line 5 continues. q is incremented to 1001, but as y is not watched by C, `propagate` is not called again.

Finally, let the solver backjump to the root by calling `backjump`(0). First, literal y is unassigned, which is not watched by C (even though its negation $\overline{y}$ is) so its watch slack is not updated. Then, $\overline{w_{998}}$ is unassigned, which is watched by C, so $C.\mathit{wslk}$ is incremented to 2. Next, $\overline{z}$ is unassigned, which is watched by C, so $C.\mathit{wslk}$ is incremented to 3. At this point, q decreased from 1001 to 998. For the remaining 998 iterations, no further adjustments to $C.\mathit{wslk}$ are needed, as none of its watches $\{y, z, w_{998}, w_{999}, w_{1000}\}$ are falsified.

3.3 Algorithm Analysis

The following two invariants underpin the soundness and completeness of our approach. Short proof sketches are available online [12].

Lemma 4 (Watch slack invariant). *The procedures* `processWatches` *(calling* `propagate`*) and* `backjump` *preserve the property*

$$C.\mathit{wslk} = \mathit{watchslack}(C, \rho^q) \tag{6}$$

Lemma 5 (Watch set invariant). *The procedures* `processWatches` *(calling* `propagate`*) and* `backjump` *preserve the property*

$$C.\mathit{wslk} < \mathit{maxcf}(C) \Rightarrow \forall \ell \in \mathit{lits}(C) \setminus \mathit{watches}(C) \colon \overline{\ell} \in \rho \tag{7}$$

for a constraint C if the argument of `backjump` *is chosen in such a way that for all constraints C where $C.\mathit{wslk} < \mathit{maxcf}(C)$, either all of its falsified watches become unassigned, or none of its non-watched literals become unassigned.*

To maintain the watch set invariant, the solver has to take care where to backjump. Withholding detail, the well-known technique of partitioning the current assignment in contiguous *decision levels* and backjumping over each level as a whole maintains the watch set invariant.

Lemma 6. *If the watch set and watch slack invariants hold, calling the procedure* `processWatches` *(calling* `propagate`*) propagates literal ℓ_i with coefficient c_i in constraint C only if it is unassigned and $\mathit{slack}(C, \rho) < c_i$, and reports that C is conflicting only if $\mathit{slack}(C, \rho) < 0$. I.e.,* `processWatches` *is sound.*

Lemma 7. *Assuming the watch set and watch slack invariant hold, if the procedure* `processWatches` *(calling* `propagate`*) returns OK, no conflicting constraint under ρ exists, and no further propagations under ρ are possible. I.e.,* `processWatches` *is complete.*

3.4 Two Optimizations

The datastructures needed for our proposed algorithm are fairly simple and should have a linear memory footprint and take (amortized) constant time for each operation. However, a performance bottleneck resides in the loops at lines 2 and 15 in `propagate`. E.g., Example 3 frequently iterates over the full size of the constraint to find new watches, even though it was clear no new watches were available. By reducing the time spent in those loops or even avoiding to enter them at all, we can improve efficiency.

First observe that when calling `propagate` because $\overline{\rho[q]}$ is watched by C, if $C.\mathit{wslk} < \mathit{maxcf}(C)$ holds at the end of the loop at line 2, all potential watches have been exhausted per the watch set invariant. Hence, when calling `propagate` for $\overline{\rho[q']}$ with $q' > q$ without backjumping over q, the loop at line 2 can be skipped. To detect this situation, we check whether $C.\mathit{wslk} + C[\mathit{idx}].\mathit{coef} < \mathit{maxcf}(C)$. If it holds, there was an earlier call to `propagate` that exited the loop at line 2 with $C.\mathit{wslk} < \mathit{maxcf}(C)$, so the loop can now be safely skipped.[6]

Next observe that, for a given constraint, any literal that is checked to become a watch by the loop at line 2, but that was not available as watch because it was falsified or already a watch (line 4 fails), can only become available as watch after a backjump occurs, since without a backjump the current assignment is only extended. Similarly, literals checked to be propagated by the loop at line 15 cannot be propagated later without a backjump occurring. To exploit this, we permanently store the indices i and j of the loops at lines 2 and 15 for each constraint (e.g., $C.i$ and $C.j$) and only reset them to 0 if a backjump happened. The latter condition is simple to check: keep a global variable (e.g., *bkjmps*) that increments by 1 at each backjump. For each constraint, check whether the global *bkjmps* matches a local copy $C.\mathit{lastbkjmp}$ that is set at each `propagate` call.

Procedure `propagateOpt` extends `propagate` with these two optimizations. Remark that as a result of these optimizations, in between backjumps, all calls to `propagateOpt` with a given constraint C, perform only $O(\mathit{size}(C))$ operations in aggregate.

4 Related Work

A PB propagation algorithm closely related to our work is that of the *Pueblo* solver [28]. It also sorts the terms of a constraint in decreasing coefficient order and checks for propagation if the slack over the watched literals of the constraint is less than the maximum coefficient of the constraint. However, except in the case of a conflicting constraint, it does not keep falsified literals as watch, as per `propagate`. It also does not update the watch slack during backjumps, as per `backjump`. Hence, it is not clear how *Pueblo* would restore the watches in the restart scenario described at the end of Example 3.[7] Also, *Pueblo* does not

[6] Note that this first optimization depends on the watch set invariant, and thus on an appropriate backjump scheme.

[7] After inquiring with the authors, the source code of *Pueblo* no longer seems available.

```
Procedure. propagateOpt(constraint $C$, integer $idx$)
  External data: watch list $wlist$, current assignment $\rho$, backjump count $bkjmps$
  Result: OK if $C$ is not falsified, otherwise CONFLICT
  if $C.lastbkjmp < bkjmps$ then
      $C.i \leftarrow 1$
      $C.j \leftarrow 1$
      $C.lastbkjmp \leftarrow bkjmps$
  if $C.wslk + C[idx].coef \geq maxcf(C)$ then
      while $C.i \leq size(C)$ and $C.wslk < maxcf(C)$ do
          $\ell \leftarrow C[C.i].lit$
          if $\overline{\ell} \notin \rho$ and $C[C.i].wflag = 0$ then
              $C[C.i].wflag = 1$
              $wlist(\ell) \leftarrow wlist(\ell) \cup \{(C, C.i)\}$
              $C.wslk \leftarrow C.wslk + C[C.i].coef$
          $C.i \leftarrow C.i + 1$
  if $C.wslk \geq maxcf(C)$ then
      $C[idx].wflag = 0$
      $wlist(C[idx].lit) \leftarrow wlist(C[idx].lit) \setminus \{(C, idx)\}$
      return OK
  if $C.wslk < 0$ then return CONFLICT
  while $C.j \leq size(C)$ and $C.wslk < C[C.j].coef$ do
      $\ell \leftarrow C[C.j].lit$
      if $\ell \notin \rho$ and $\overline{\ell} \notin \rho$ then $\rho$.push($\ell$)
      $C.j \leftarrow C.j + 1$
  return OK
```

implement the optimizations described in Sect. 3.4, and does not store the index of a watched literal of a constraint in the watch lists, which might lead to a linear lookup overhead or require a cache-inefficient associative array.

Before *Pueblo*, work on the *Galena* solver [6] also prompted PB propagation investigation. It uses a watched propagation scheme where the number of watches of a constraint depends on a dynamic maximum coefficient a_{max} of the literals currently not assigned to true. This minimizes the number of watched literals, but according to [27], two thirds of the run time of the *Galena* propagation procedure is spent updating a_{max} for each constraint. Because of this, it was proposed to keep a_{max} fixed to the highest coefficient (i.e., $maxcf(C)$), but *Galena* eventually settled on a three-tiered approach with watched propagation only for clauses and cardinality constraints, and counter propagation for general PB constraints [6].

The more recent *Sat4J* uses this three-tiered approach by default, but provides the option to enable a less efficient watched propagation [18].

Finally, the *RoundingSat* solver [14] implements a watched propagation algorithm which, as in our approach, uses a static maximum coefficient to calculate the number of needed watches and keeps watching falsified literals, but swaps watched literals to the front of the constraint [26]. This makes calculating the watch slack after every call relatively efficient, as only the watched literals in the front of the constraint need to be iterated over, rendering the update of the watch slack during backjumps obsolete. As the watch swaps alter the order of the literals of the constraint, the index of a watched literal cannot be stored in the watchlist, as in our approach, and is recalculated during watch slack calculation. To check for propagating literals, *RoundingSat* again always iterates over all watched literals. However, the number of watches of a PB constraint can grow linearly in the size of the constraint, which leads to a potentially large overhead for constraints that require lots of watches.

On the CP side, to the best of our knowledge, the constraint in the global constraint catalog most closely related to PB constraints is *sum_set* [29], which constrains an integer variable V to take the sum of a variable subset of a set of values. In the special case where V is constrained only by one fixed bound, sum_set is equivalent to a PB constraint. The propagator for sum_set in the CP solver *Gecode* [15] relies on counter propagation, though the comparison is not fully fair as not only literals have to be propagated, but bounds on V as well.

5 Experimental Evaluation

To experimentally evaluate our proposed propagation algorithm, we implemented it in the *RoundingSat* PB solver [26]. Source code, a binary, and raw experimental data are available online [12]. As hardware we used AMD Opteron 6238 nodes having 6 cores and 16 GiB of memory each. Each run was executed as a single thread on a node with a 5000 s timeout limit.

To make a sufficiently broad comparison, we present experiments on instances from the linear small coefficient decision and optimization tracks from the most recent PB competition [24], referred to as PB16dec and PB16opt. Additionally, we investigate 0-1 integer linear programming instances from the MIPLIB libraries [1,3,16,17,21,22]. Since these sets contain few decision instances, we also created decision versions of the optimization problems. For this, we constructed a first instance by replacing the objective function f with a constraint stating that f should be at least the best known value, and a second where f should be strictly better. As *RoundingSat* can currently only deal with integer coefficients of magnitude at most 10^9, some of the instances were rescaled and rounded. We refer to the corresponding MIPLIB decision and optimization problems as MIPLIBdec and MIPLIBopt. These instances are available online [11].

5.1 Two Optimizations to Watched PB Propagation

Let's start with a simple question: how effective are the two optimizations described in Sect. 3.4? For this, we implemented in *RoundingSat* watched propagation per Procedure `propagate` (*watch*) and per Procedure `propagateOpt` (*watch-opt*), and compare the propagation speed defined as the total propagations performed divided by the solve time. As *watch* and *watch-opt* do not differ in the order in which propagations happen, the runs for both *watch* and *watch-opt* have the same conflict and decision counts and any difference in propagation speed is solely due to algorithmic efficiency. Figure 1 plots the result for the instances that were solved by both *watch* and *watch-opt* within resource limits and took at least 1 s to solve. The result is clear: the optimizations can increase the propagation speed by an order of magnitude and never incur significant overhead.

5.2 Expensive Backjumps?

One advantage of watched propagation in SAT solvers is that no work needs to be done during backjumps, a feature preserved by the original propagation implementation of *RoundingSat*. Our approach updates the watch slack during backjumps, though only for those constraints C that have falsified watches, which only happens if $C.wslk < maxcf(C)$. Figure 2 plots the number of times *watch-opt* looked up a constraint when backjumping over a falsified watched literal (line 6 in `backjump` and 11 in `processWatches`) versus the number of times it looked up a constraint during propagation of a watch (lines 7 and 8 in `processWatches`), for instances solved within resource limits.

Backjump lookups happen frequently, but never more than propagation lookups. Often, backjump lookups happen significantly less than propagation lookups, up to two orders of magnitude. The median number of backjump watch lookups is also less than half the median of propagation watch lookups. As backjump lookups perform few operations compared to propagation lookups, the resulting overhead does not seem to induce a performance bottleneck.

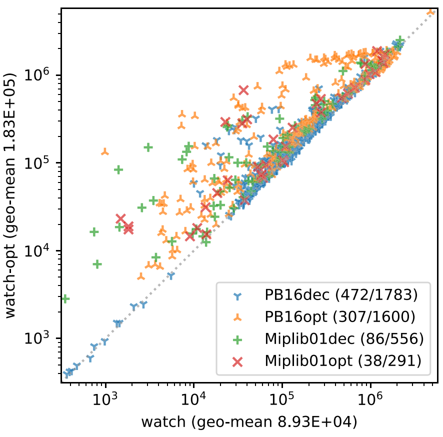

Fig. 1. Propagations per second for *watch* and *watch-opt*.

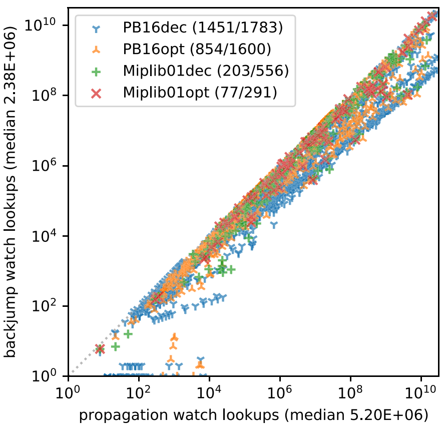

Fig. 2. Watch lookups for *watch-opt*.

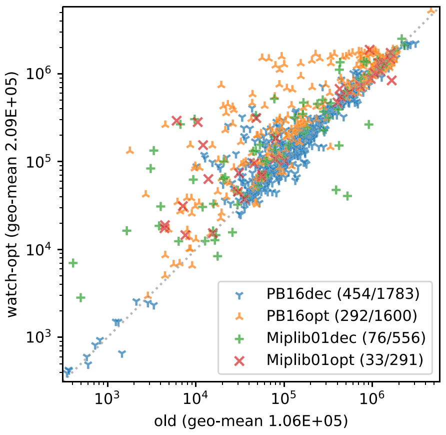

Fig. 3. Propagations per second for *old* and *watch-opt*.

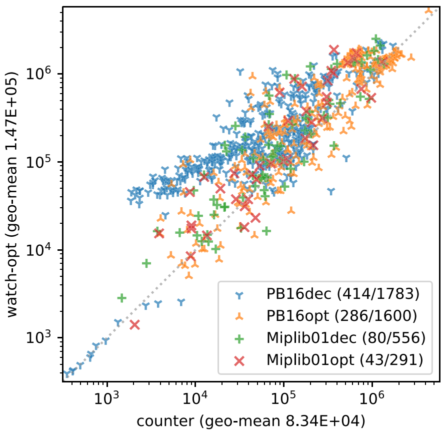

Fig. 4. Propagations per second for *counter* and *watch-opt*.

5.3 Performance Evaluation

To evaluate the performance of our approach, we compare *watch-opt* to:

- *counter*: an implementation of PB counter propagation (see Sect. 2.2)
- *old*: the original propagation algorithm of *RoundingSat* (see Sect. 4)
- *counter-cc*: *counter*, but **c**lauses and **c**ardinality constraints are handled with specialized watched propagation routines – the three-tiered approach default in *Sat4J* (see Sect. 4)
- *old-cc*: three-tiered *old* with the same specialized routines
- *watch-opt-cc*: three-tiered *watch-opt* with the same specialized routines

Figures 3, 4, 5 and 6, compare the propagation speed of *watch-opt* to the above alternatives, based on the instances succesfully solved by the compared approaches within resource limits and taking at least 1 s to solve. Table 1 presents the total number of succesfully solved instances by each approach.

Often, the propagation speed of *watch-opt* is orders of magnitude faster than of *old* and *counter*, with the reverse being true only infrequently. This translates to significantly more solved instances compared to *old* and *counter*. The specialized propagation for clauses and cardinality constraints improves performance in general, with most *-cc* configurations solving more instances than their counterparts. *watch-opt-cc* solves the most instances overall, while *counter-cc* seems to profit most from the specialized routines, almost fully closing the gap with *watch-opt-cc*. The propagation speed plots in Figs. 5 and 6 tell a similar tale: *old-cc* propagates significantly slower than *watch-opt-cc*, but it becomes harder to judge that *watch-opt-cc* propagates faster. The geometric means of their propagation speed in Fig. 6 still give the edge to *watch-opt-cc*.

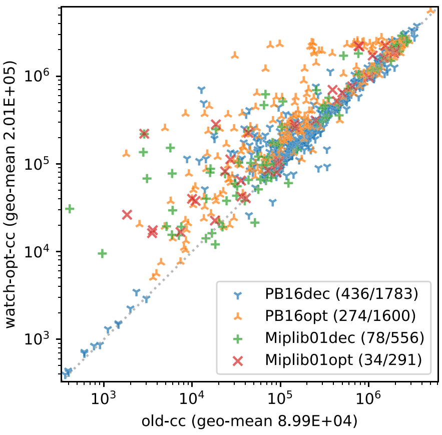

Fig. 5. Propagations per second for *old-cc* and *watch-opt-cc*.

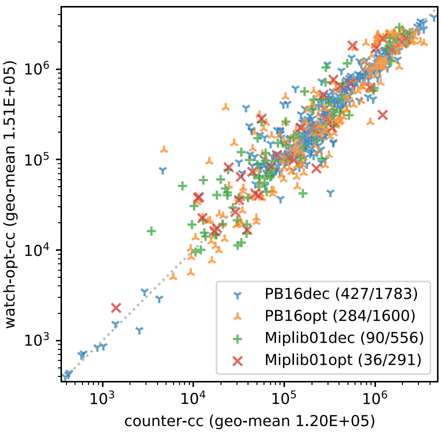

Fig. 6. Propagations per second for *counter-cc* and *watch-opt-cc*.

Table 1. Solved instance counts for different propagation implementations

	old	*counter*	*watch-opt*	*old-cc*	*counter-cc*	*watch-opt-cc*
PB16dec (1783)	1429	1385	1451	1444	1456	**1472**
MIPLIBdec (556)	182	196	203	187	204	**205**
PB16opt (1600)	820	846	854	825	**862**	854
MIPLIBopt (291)	69	76	77	71	75	**79**

To explain the relative difference between *old*/*old-cc* and *counter*/*counter-cc*, it is useful to characterize when *counter* and *old* accrue the most overhead. A counter algorithm induces most overhead for constraints with low watch count as continually updating the high slacks for these constraints is often unnecessary. Inversely, *old* incurs more overhead for constraints that have a relatively high number of watches, as its eager recalculation of watch indices, watch slacks, and propagating watches, are linear operations in the number of watches. Since clauses and low-degree cardinality constraints are frequently generated constraints with low watch counts, this can explain why *counter* profits a lot more from the specialized propagation routines than *old*.

We conclude that *watch-opt* is indeed more efficient than its *counter* counterpart. However, adding specialized clause and cardinality constraint propagation into the mix strongly diminishes its advantage – *counter-cc*, the *Sat4J* default approach, is definitely a close second.

6 Conclusion

We present an optimized watched propagation algorithm for PB or 0-1 integer linear constraints. Our experiments indicate it is more efficient than counter propagation used by *Sat4J* and the watched propagation used by *RoundingSat*. Hence, our approach seems a good candidate to replace PB counter propagation with PB watched propagation, though the performance gains are moderate in the three-tiered setting. Nonetheless, the results are sufficiently convincing to consider *watch-opt-cc* as a new default propagation algorithm for *RoundingSat*.

An interesting avenue to speed up PB propagation would be to pinpoint which PB constraints propagate more efficiently with a counter approach and which favor the watched approach. Maybe those constraints which most of the time have a relatively large number of watched literals are better off with the counting approach? Other future work may reconsider the idea of *Galena*: track the largest coefficient of non-true literals to reduce the number of watches for a constraint. Our work can also prove useful to improve CP propagators for constraints closely related to PB constraints, such as the sum_set constraint. Finally, the order in which constraints propagate strongly influences what a conflict-driven solver will learn. Prioritizing certain types of constraints during propagation may yield better learned constraints.

Acknowledgments. The Swedish National Infrastructure for Computing (SNIC) at the High Performance Computing Center North (HPC2N) at Umeå University provided computational resources. The author is supported by the Swedish Research Council grant 2016-00782.

We are grateful to Emir Demirovic, Jan Elffers, Stephan Gocht, Daniel Le Berre and Jakob Nordström for discussions on PB propagation.

References

1. Achterberg, T., Koch, T., Martin, A.: MIPLIB 2003. Oper. Res. Lett. **34**(4), 361–372 (2006). https://doi.org/10.1016/j.orl.2005.07.009, http://www.zib.de/Publications/abstracts/ZR-05-28/
2. Bayardo Jr., R.J., Schrag, R.: Using CSP look-back techniques to solve real-world SAT instances. In: Proceedings of the 14th National Conference on Artificial Intelligence (AAAI 1997), pp. 203–208 (1997)
3. Bixby, R., Ceria, S., McZeal, C., Savelsbergh, M.: An updated mixed integer programming library: MIPLIB 3.0 (1998)
4. Blake, A.: Canonical expressions in Boolean algebra. Ph.D. thesis, University of Chicago (1937)
5. Buss, S., Nordström, J.: Proof complexity and SAT solving. In: Handbook of Satisfiability, 2nd edn. (2020, to appear). Draft version. http://www.csc.kth.se/~jakobn/research/
6. Chai, D., Kuehlmann, A.: A fast pseudo-Boolean constraint solver. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **24**(3), 305–317 (2005). Preliminary version in DAC 2003
7. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC 1971), pp. 151–158 (1971)
8. Cook, W., Coullard, C.R., Turán, G.: On the complexity of cutting-plane proofs. Discrete Appl. Math. **18**(1), 25–38 (1987)

9. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Commun. ACM **5**(7), 394–397 (1962)
10. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**(3), 201–215 (1960)
11. Devriendt, J.: Miplib 0–1 instances in OPB format (2020). https://doi.org/10.5281/zenodo.3870965
12. Devriendt, J.: Online Repository for "Watched Propagation of 0–1 Integer Linear Constraints" (2020). https://doi.org/10.5281/zenodo.3952444
13. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
14. Elffers, J., Nordström, J.: Divide and conquer: towards faster pseudo-Boolean solving. In: Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI 2018), pp. 1291–1299 (2018)
15. Gecode: Generic constraint development environment. https://www.gecode.org/
16. Gleixner, A., et al.: MIPLIB 2017: data-driven compilation of the 6th mixed-integer programming library. Technical report, Optimization Online (2019). http://www.optimization-online.org/DB_HTML/2019/07/7285.html
17. Koch, T., et al.: MIPLIB 2010. Math. Programm. Comput. **3**(2), 103–163 (2011). https://doi.org/10.1007/s12532-011-0025-9, http://mpc.zib.de/index.php/MPC/article/view/56/28
18. Le Berre, D., Parrain, A.: The Sat4j library, release 2.2. J. Satisfiability Boolean Model. Comput. **7**, 59–64 (2010)
19. Levin, L.A.: Universal sequential search problems. Problemy peredachi informatsii **9**(3), 115–116 (1973). (in Russian). http://mi.mathnet.ru/ppi914
20. Marques-Silva, J.P., Sakallah, K.A.: GRASP: a search algorithm for propositional satisfiability. IEEE Trans. Comput. **48**(5), 506–521 (1999). Preliminary version in ICCAD 1996
21. MIPLIB 2.0 (1996). http://miplib2010.zib.de/miplib2/miplib2.html
22. MIPLIB 2017 (2018). http://miplib.zib.de
23. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference (DAC 2001), pp. 530–535 (2001)
24. Pseudo-Boolean competition 2016 (2016). http://www.cril.univ-artois.fr/PB16/
25. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12**(1), 23–41 (1965)
26. RoundingSat. https://gitlab.com/miao_research/roundingsat
27. Sheini, H.M., Sakallah, K.A.: Pueblo: a modern pseudo-Boolean SAT solver. In: Proceedings of the Design, Automation and Test in Europe Conference (DATE 2005), pp. 684–685 (2005)
28. Sheini, H.M., Sakallah, K.A.: Pueblo: a hybrid pseudo-Boolean SAT solver. J. Satisfiability Boolean Model. Comput. **2**(1–4), 165–189 (2006). Preliminary version in DATE 2005
29. Global constraint catalog: sum_set. https://sofdem.github.io/gccat/gccat/Csum_set.html
30. Zhang, H., Stickel, M.: Implementing the Davis-Putnam method. J. Autom. Reasoning **24**(1), 277–296 (2000). https://doi.org/10.1023/A:1006351428454, https://doi.org/10.1023/A:1006351428454