A Generalized Two-watched-literal Scheme in a mixed Boolean and Non-linear Arithmetic Constraint Solver[⋆]

Tino Teige[1], Christian Herde[1], Martin Fränzle[1], Natalia Kalinnik[2], and Andreas Eggers[1]

[1] University of Oldenburg, Germany
{teige|herde|fraenzle|eggers}@informatik.uni-oldenburg.de
[2] University of Freiburg, Germany
kalinnik@informatik.uni-freiburg.de

Abstract. In its combination with conflict-driven clause learning the two-watched-literal scheme led to enormous performance gains in propositional SAT solving. The idea of this approach is to accelerate the deduction phase of a SAT solver by saving a high number of unnecessary and expensive computation steps originating in visits of indefinite clauses. In this paper we give a detailed explanation of the generalized watch scheme, called *two-watched-atom scheme*, implemented in our interval-based constraint solver HySAT, the latter being a solver for mixed Boolean and non-linear arithmetic constraint formulae. As opposed to the purely Boolean setting, the more general form of atomic formulae to be watched in our solver necessitates an extension of the original scheme and calls for a careful design of the data structures employed in the implementation. We present experimental results to demonstrate the speed-up obtained by the proposed scheme.

1 Introduction

Concerning performance of SAT solvers based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [DP60,DLL62], one of the most important improvements in recent years can be attributed to a novel implementation of unit propagation, introduced with the mChaff SAT solver [MMZ$^+$01]. 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 work has been partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, www.avacs.org).

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, referred to as *two-watched-literal scheme* (2WLS), has been shown to achieve significant performance gains in interplay with learning and non-chronological backtracking, especially on hard SAT instances [MMZ$^+$01].

In this paper we explain in detail the generalized two-watched-literal scheme of our interval-based constraint solver HySAT[3] [FHR$^+$07]. HySAT tackles the in general undecidable problem of solving mixed Boolean and non-linear arithmetic constraint formulae over the reals and therefore can be employed, e.g., for the reachability analysis of hybrid systems. In contrast to the propositional case, more general literals, called atoms, are watched in the HySAT framework. An *atom* can contain up to three variables connected by a relational and an arithmetic operator, e.g. $x = y \cdot z$. Furthermore, HySAT manipulates interval valuations during its proof search s.t. the interpretation (truth value) of an atom can be *inconclusive* (in addition to true and false) under an interval valuation. In our more general case we thus have to deal with behavior different from propositional SAT solving, i.e.

1. If the interval of a variable x has changed, we do not know whether an atom a containing x becomes false without evaluating the new truth value of a. So, a watched atom in a clause can potentially become *true* or *false*, or still remain *inconclusive*.
2. In contrast to the Boolean case, where clauses are normalized to contain at most one literal per variable, it can happen that both watches become false because of a new interval for one variable: Let $x = 3 \cdot z$ and $y = \sin(x)$ be the currently watched atoms in a clause. Then an interval contraction for x could falsify both atoms.
3. If the watched atom $x = y + z$ becomes false, say implied by a contraction of y's interval, we have to efficiently inform variables x and z that their watch sets should be updated as the atom $x = y + z$ is no longer watched.

The aim of the paper is to shed light on the concepts and ideas behind our implementation and to demonstrate their impact by means of experimental results.

Related work. There are various successful attempts to adapt the two-watched-literal scheme from the Boolean domain (originated in [MMZ$^+$01]) to a more general case of problem solving. In [CK03] and [FH03] a generalization of the two-watched-literal scheme for pseudo-Boolean constraints was proposed. In [GJM06] the idea of a watch scheme is lifted to "element" (of an array) and "table" constraints. These three approaches just handle discrete (Boolean or integer)

[3] An HySAT executable, the tool documentation, and some benchmarks can be found on `http://hysat.informatik.uni-oldenburg.de`.

variables, respectively. However, the 2WLS was also applied in constraint satisfaction problems (CSPs) with potentially continuous domains. A main intention here, aside from accelerating the clause evaluation and propagation, is to battle against the massive amount of learned clauses (aka *nogoods*) and the resulting memory problems within solving a CSP. The literal watching is exploited for a more efficient recording and managing in [KB03,RCOJ06] as well as for minimizing nogoods in [LSTV07].

In contrast to our watch scheme, all of the aforementioned approaches just support watching of "simple" literals, i.e. a relation between *one* variable and a value. In this paper, we consider more complex literals, i.e. we actively watch atoms containing up to *three* variables, a relational as well as an arithmetic operator and evaluate these by means of interval arithmetic.

Structure of the paper. In section 2 we briefly give some relevant definitions, while section 3 presents a detailed description of our generalized watch scheme and provides benchmark results to demonstrate the speed-up obtained by the proposed scheme. The paper concludes with section 4.

2 Preliminaries

In [FHR$^+$07] we introduced the iSAT algorithm (implemented in the tool HySAT) for solving non-linear arithmetic constraint systems with complex Boolean structure. This algorithm is a tight integration of SAT solving based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [DP60,DLL62] and interval constraint propagation (ICP, cf. [BG06] for an extensive survey) enriched by enhancements like conflict-driven learning and non-chronological backtracking. A very detailed theoretical description of the algorithm and benchmark results can be found in [FHR$^+$07]. Our novel approach outperforms the classical lazy theorem approach of ABSOLVER [BPT07], which —to the best of our knowledge— is the only other tool addressing mixed Boolean and non-linear arithmetic satisfiability problems. Due to space limitations we cannot recall the iSAT algorithm here. The reader is referred to the original paper, in particular to the example on pages 217–219.

The input constraint formulae of HySAT are arbitrary Boolean combinations of non-linear arithmetic constraints over the reals. By the front-end of our constraint solver, these constraint formulae are rewritten to linearly sized, equi-satisfiable formulae in conjunctive normal form, with atomic propositions and arithmetic constraints confined to a form resembling three-address code. This rewriting is based on the standard mechanism of introducing auxiliary variables for the values of arithmetic sub-expressions and of logical sub-formulae. Thus, the *internal* syntax[4] of constraint formulae is as follows:

$$\begin{aligned} \mathit{formula} &::= \{\mathit{clause} \wedge\}^* \mathit{clause} \\ \mathit{clause} &::= (\{\mathit{atom} \vee\}^* \mathit{atom}) \end{aligned}$$

[4] For examples of the user-level syntax, consult the benchmark files on the HySAT website `http://hysat.informatik.uni-oldenburg.de/`.

$$\begin{aligned}
\textit{atom} &::= \textit{bound} \mid \textit{equation} \\
\textit{bound} &::= \textit{variable relation rational\_const} \\
\textit{relation} &::= < \mid \leq \mid = \mid \geq \mid > \\
\textit{equation} &::= \textit{pair} \mid \textit{triplet} \\
\textit{pair} &::= \textit{variable} = \textit{uop variable} \\
\textit{triplet} &::= \textit{variable} = \textit{variable bop variable}
\end{aligned}$$

where *uop* and *bop* are unary and binary operation symbols, resp., including $+$, $-$, $\times$, sin, etc., and *rational_const* ranges over the rational constants.

The definition of satisfaction is standard: a constraint formula ϕ is satisfied by a valuation of its variables iff all its clauses are satisfied, that is, iff at least one atom is satisfied in any clause. Satisfaction of atoms is wrt. the standard interpretation of the arithmetic operators and the ordering relations over the reals. Instead of real-valued valuations of variables, our constraint solving algorithm manipulates interval-valued valuations $\rho : V \to \mathbb{I}_\mathbb{R}$, where $\mathbb{I}_\mathbb{R}$ is the set of convex subsets of $\mathbb{R}$.[5] If both ρ' and ρ are interval valuations then ρ' is called a *refinement* of ρ iff $\rho'(v) \subseteq \rho(v)$ for each variable v. Let x, y be variables, ρ be an interval valuation, and $\circ$ be a binary operation. Then $\rho(x \circ y)$ denotes the *interval hull* of $\rho(x)\hat{\circ}\rho(y)$ (i.e. the smallest enclosing interval which is representable by machine arithmetic), where the operator $\hat{\circ}$ corresponds to $\circ$ but is canonically lifted to sets. Analogously for unary operators.

We say that an atom a is *inconsistent* under an interval valuation ρ iff

$$\begin{array}{ll}
\rho(x) \cap \{u \mid u \in \mathbb{R}, u \sim c\} = \emptyset & \text{if } \ a = (x \sim c), \\
\rho(x) \cap \rho(\circ y) = \emptyset & \text{if } \ a = (x = \circ y), \\
\rho(x) \cap \rho(y \circ z) = \emptyset & \text{if } \ a = (x = y \circ z).
\end{array}$$

Otherwise a is *consistent* under ρ. For explaining the watch scheme we do not need the definition of interval satisfaction. It is sufficient to talk about atoms, which are still consistent (i.e., either *true* or *inconclusive*). For the issue of interval satisfaction the reader is referred to [FHR$^+$07] subsection 4.5. Note that deciding (in)consistency of an atom under an interval valuation is straightforward.

3 Generalized two-watched-literal scheme

In this section we describe in detail the generalized watch scheme of HySAT, called *two-watched-atom scheme* (2WAS), during solving a formula φ of syntax introduced in section 2. Wlog. and as stated by the syntax, we assume that φ contains at least one clause and each clause at least one atom, otherwise φ is trivially true or false, respectively.

[5] Note that we also support discrete domain (integer and Boolean) variables. To this end it suffices to clip the interval of integers variables accordingly, such that $[-3.4, 6.0)$ becomes $[-3, 5]$, for example. The Boolean domain is represented by $\mathbb{B} = [0, 1] \subset \mathbb{Z}$.

In subsection 3.1 we consider all possible cases which can occur when evaluating a clause with two watched atoms (issues 1 and 2 of section 1). Subsection 3.2 deals with the implementation of this watch scheme in our tool HySAT. An important issue here is the efficient manipulation of the watch lists (issue 3 of section 1).

3.1 Clause evaluation and finding new watches

Let $c = (a_0 \vee \ldots \vee a_n) \in \varphi$, $n \geq 0$, be a clause. As in the propositional case we call a clause c *unit* under an interval valuation ρ if exactly n distinct atoms in c are inconsistent under ρ.

Before solving φ, we set the watched atoms for each clause c. If $n = 0$ then c is initially *unit*. Otherwise, we (randomly) select two atoms $a_i, a_j \in c$ with $i \neq j$ as watches of c. Whenever interval contraction for some variable x causes a visit of a clause c, we potentially have to evaluate both watched atoms a_i, a_j of c, since x could occur in a_i as well as in a_j. We distinguish three cases:

1. Both a_i and a_j remain consistent,
2. exactly one of the atoms a_i and a_j becomes inconsistent, or
3. both a_i and a_j become inconsistent.

In case 1, no action is required. In cases 2 or 3 we start searching for new watches. For case 2, wlog., let a_i be the inconsistent atom. If we find a replacement for a_i, i.e. a consistent atom $a \in c$ with $a \notin \{a_i, a_j\}$, then the atoms a and a_j are watched in c. Otherwise, we propagate the (consistent) atom a_j since c is unit.

For case 3, we have to distinguish three subcases:

a) If we succeed in finding two consistent, unwatched atoms a and a' then we can use these as replacements for a_i and a_j.
b) If c contains only one other consistent atom $a \notin \{a_i, a_j\}$ then we have to propagate the (consistent) atom a since c is unit.
c) Otherwise all atoms in c are inconsistent, i.e. the clause is conflicting. To notify the solver about the conflict we propagate any of c's atoms. As soon as HySAT tries to assert this atom it will end up with a variable having an empty range, thus making the conflict apparent.

3.2 Efficient implementation

In subsection 3.1 we have examined which cases can occur when evaluating a clause. However, we have not explained so far how we store the information which atoms are watched in which clauses. We have to access that information efficiently when a new interval border for a variable is propagated. Additionally, we have to update this information efficiently when watched atoms are replaced. Such technical issues are of vital importance wrt. an overall efficient implementation of a solver (as already known for the propositional case).

Although our generalized watch scheme looks very similar to those of state-of-the-art propositional SAT solvers like, e.g., MiniSat [ES03] and MiraXT [LSB07],

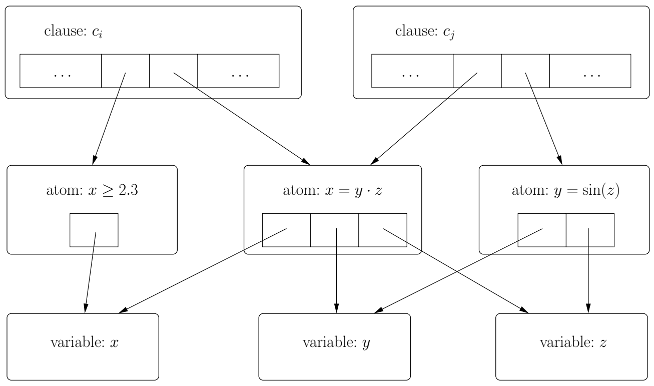

Fig. 1. The data-structure of HySAT: Arrows denote pointers to the corresponding objects.

a precise discussion for the more general framework reveals some subtleties not known from the propositional case as mentioned in section 1. While subsection 3.1 resolves issues 1 and 2, here we concentrate on efficient data-structures for efficiently manipulating the watch lists (issue 3).

Data-structure. We represent variables, atoms (i.e., bounds and equations), and clauses as *objects*[6] in the programming-language C++. Our algorithm, then, handles pointers to these objects. Each clause possesses a vector of pointers to its atoms, while each atom knows the pointers to its variables (cf. Fig. 1). The atoms in the vector of (pointers to) atoms of a clause on the *first* and *second* position are watched, as, e.g., implemented in MiniSat [ES03]. Then, setting new watches is done by swapping entries of the vector.

In the sequel, we discuss how to extend our data-structure in order to inform the variables that atoms become watched or are no longer watched. A naive approach could be —easily adapted from propositional SAT engines— to introduce for each variable x a so called *watch list*, i.e. a list $\langle c_0, \ldots, c_k \rangle$ of (pointers to) clauses in which an atom containing x is currently watched. Whenever a newly deduced interval, more precisely a newly deduced interval border, for x is propagated, the naive algorithm visits and evaluates the clauses $c_0, \ldots, c_k$ (as mentioned in subsection 3.1). Note that the information about such deduced interval borders are stored in a queue, called *implication queue* as in the propositional case. The entries of the implication queue, i.e. the interval borders, are propagated consecutively. By this sequential process we avoid nesting within the propagation phase similarly as in Boolean SAT solving.

When the evaluation of clause c_j (currently stored on position i in the watch list of variable x) results in the situation that a watched atom $a \in c_j$ containing

[6] We decide for a clear and –as far as possible– modular data-structure rather than a low-level implementation (as done for state-of-the-art SAT solver) due to the more complex implementation of the algorithm.

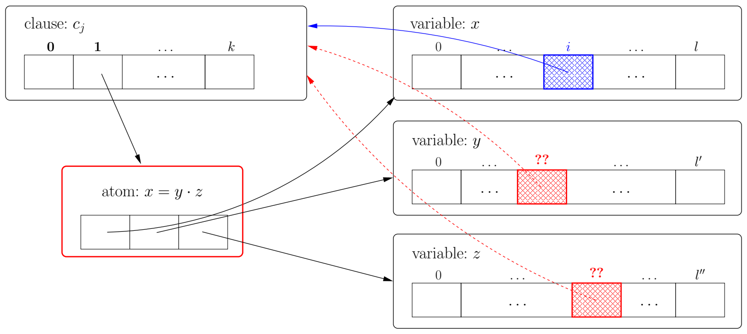

Fig. 2. A naive watch scheme: To propagate a new interval border for variable x we traverse the watch list of x. When visiting clause c_j we recognize that the watched atom $a = (x = y \cdot z)$ becomes inconsistent. Thus, we have to replace a by a new watch in c_j and remove the entries for c_j from the watch lists of x, y, and z. Since we do not know the corresponding indices for the watch lists of y and z a priori, we have to perform a time-consuming list search.

x is inconsistent under the current interval valuation and a replacement for a was found in c_j then we have to remove the pointer to c_j from the watch list of x (provided that the new watched atoms do not contain x). This is easily and efficiently feasible (in constant time) by copying the entry at the last position of the watch list to position i, followed by removing the last field of the watch list, i.e.

$$\langle c_l, \ldots, c_j, \ldots, c_m, c_n \rangle \leadsto \langle c_l, \ldots, c_n, \ldots, c_m, c_n \rangle \leadsto \langle c_l, \ldots, c_n, \ldots, c_m \rangle.$$

However, the inconsistent atom a potentially contains other variables $v \neq x$, e.g. if $a = (x = y \cdot z)$. We also have to update the watch lists of the variables $v \neq x$. Since we do not know on which position the clause c_j is stored in the watch lists of the variables $v \neq x$, a naive solution might be to go through the whole watch lists until we found the corresponding entries. This situation is depicted in Fig. 2. Our aim is to avoid this naive approach, since such watch list manipulations are one of the main operations and the watch lists can become very large. (HySAT tackles formulae with some thousands of variables and clauses, cf. [FHR$^+$07] section 5 and the HySAT website[7].)

Two-level-watch scheme. Our solution to avoid iterating through the whole watch lists when a watched atom becomes unwatched is as follows. We introduce a *two-level-watch scheme*, i.e. variables are watched in atoms and atoms are watched in clauses. Therefore, each variable x handles watch lists of pointers to currently watched atoms containing x. Each atom manages a watch list of pointers to clauses in which it is watched. Clearly, by this data-structure, we maintain a low-cost access to the clauses to be visited when a new interval border for x

[7] `http://hysat.informatik.uni-oldenburg.de/`

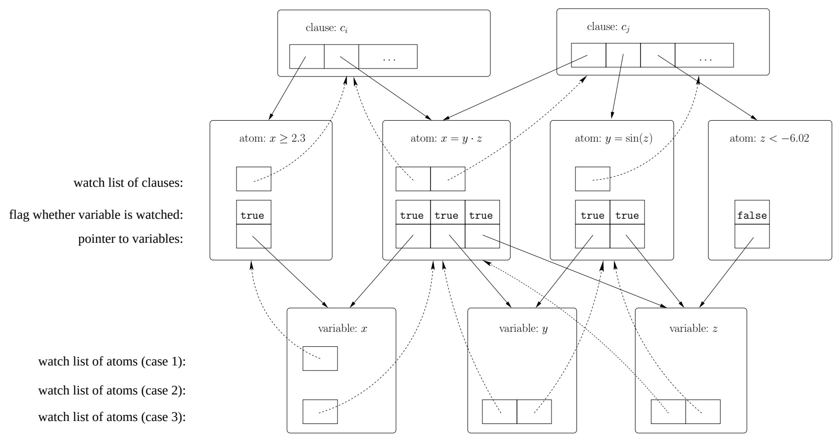

Fig. 3. The two-level-watch scheme of HySAT: The atoms $x \geq 2.3$ and $x = y \cdot z$ are watched in clause c_i. The equation $x = y \cdot z$ is also watched in clause c_j, together with $y = \sin(z)$. The corresponding data-structure is depicted, where dashed arrows denote pointers in watch lists. The atom $z < -6.02$ is not watched in c_j since it is not on the first or second position in the vector of atoms in c_j.

is propagated. For the other way round, i.e. an atom a becomes unwatched in a clause, we update the watch list of a but we do *not* update the watch lists of the variables y in a immediately but in a *lazy* fashion. I.e., we refresh the watch lists of such variables y when a new interval border for y is propagated. Then, we have the position in the watch list of the non-watched atom a at hand and can delete the entry for a as mentioned above. This procedure avoids naive searching for watch list entries.

In our implementation, each variable x has *three* watch lists of atoms, namely one list of atoms a for each of the following cases:

1. Only a new *upper* interval border can make a inconsistent, i.e. a is a lower bound of form $x \sim c$ with $\sim \in \{\geq, >\}$.
2. Only a new *lower* interval border can make a inconsistent, i.e. a is an upper bound of form $x \sim c$ with $\sim \in \{<, \leq\}$.
3. Each new interval border can make a inconsistent, i.e. a is an equation or a bound of form $x = c$.

Since the implication queue contains only information on newly deduced interval borders, by using this data-structure we save pointless visits to clauses for which no new watch is needed, since the currently watched atoms cannot be inconsistent under the new interval valuation.

To serve the purpose of manipulating the watch lists of the variables in a lazy fashion, each atom a owns Boolean flags w_x for all its variables x, indicating whether x is watched in a. The extended data-structure is depicted in Fig. 3.

The algorithmic details for manipulating the data-structure are presented in Alg. 1. First, the algorithm determines the corresponding watch lists of variable

Algorithm 1 Two-watched-atom scheme of HySAT

```
In: a new interval border $x \sim c$ with $\sim \in \{<, \leq, \geq, >\}$ (from the implication queue) to
    be propagated.
 1: Let $WL_1, WL_2$, and $WL_3$ be the watch lists of variable $x$ of case 1, 2 and 3,
    respectively.
 2: if $\sim \in \{<, \leq\}$ then
 3:   $W = \{WL_1, WL_3\}$.
 4: else
 5:   $W = \{WL_2, WL_3\}$.
 6: end if
 7: for all $WL \in W$ do
 8:   for $i = 0$ to $size(WL) - 1$ do
 9:     $a = WL[i]$ is the current atom.
10:     Let $WL_a$ be the watch list of atom $a$.
11:     if $WL_a$ is empty then
12:       Remove entry at position $i$ from $WL$.
13:       for all flags $w_y$ of $a$ with $y = x$ do
14:         $w_y = \texttt{false}$.
15:       end for
16:     else if atom $a$ evaluates to inconsistent then
17:       for $j = 0$ to $size(WL_a) - 1$ do
18:         $cl = WL_a[j]$ is the current clause.
19:         if $cl$ is unit then
20:           continue.
21:         end if
22:         if replacement $a'$ for $a$ was found in $cl$ then
23:           Remove entry at position $j$ from $WL_a$.
24:           Add $cl$ to watch list $WL_{a'}$ of $a'$.
25:           for all variables $z$ of $a'$ with flag $w'_z == \texttt{false}$ do
26:             Add $a'$ to the corresponding watch list of $z$.
27:             Set all flags $w'_z = \texttt{true}$.
28:           end for
29:         end if
30:       end for
31:     end if
32:   end for
33: end for
```

x for which the interval border $x \sim c$ can violate watched atoms. Then, consecutively, each atom a of the watch lists will be visited. In a second step the clause-watch-list WL_a of atom a is traversed. In case WL_a is empty, we know that a is no longer watched in any clause. We remove the entry for a from x's watch list (in constant time as mentioned above) and set each flag w_y belonging to x to false. Otherwise, a is still watched in some clauses. If a becomes inconsistent under the new interval valuation, we cannot use a as a watched atom any longer. Therefore, we search for new watches in each clause in which a is watched. If we find a new watch a' in clause cl, we efficiently delete the entry for cl in WL_a, and add cl to the watch list of a'. Here, the flags of a' immediately indicate which variables are still watched in a'. Only if a variable z is not yet watched in a', we add the atom a' to the corresponding watch list of z and update the flag. Here, we do not update the watch lists of the variables y of the non-watched atom a in order to avoid stepping through all watch lists of that variables y. We postpone this action until new interval borders are propagated for the variables y: Since WL_a is then empty, lines 11–15 ensure the efficient update of the watch lists of the variables y.

Note that we adopt the approach used in many SAT solvers to not change the watch lists when a clause becomes or is unit. For each unit clause c, we store the decision level on which c became unit, and reset that information when c becomes non-unit again after backtracking. In the latter case, the watched atoms of c (on the first and second position in the vector of atoms of c) are always valid watches, i.e. are consistent. Thus, our data-structure is *backtrack-stable*, i.e. we save searching for new watched atoms after backtracking.

3.3 Experimental results

In order to demonstrate the potential of the generalized two-watched-literal scheme approach, we compare the performance of HySAT to a stripped version thereof, where the two-watched-atom scheme is disabled (but all other optimizations, like conflict-driven learning and backjumping, remain operational). I.e., the stripped down version always visits and evaluates a clause c if for one of the variables occurring in c, a new interval bound was assigned. All benchmarks were performed on an 1.83 GHz Intel Core 2 Duo machine with 1 GByte physical memory running Linux.

All the formulae (of the internal syntax, cf. section 2) were randomly generated[8] and are grouped. The characteristics for each group of formulae are listed in table 1, where also the runtimes and numbers of clause evaluations on average are mentioned for both versions of the solver as well as their ratios. The results for all benchmarks are depicted in Fig. 4. These experiments consistently show that the two-watched scheme yields substantial speed-ups also for the more general

[8] Given the characteristics of a formula (number of variables and clauses as well as the clause size), the types (bound or equation), variables, and, if applicable, arithmetic operators and rational constants of the atoms in clauses of the formula are chosen by uniform distribution.

group	♯var	♯cl	cl-size	runtime / ♯cl-eval		speed-up	♯cl-eval down
				2WAS on	2WAS off		
1	30	100	20	0.2s / 12,287.6	0.4 / 72,736.6	2.0	5.9
2	100	300	40	0.3s / 14,859.1	0.9s / 198,532.4	3.0	13.4
3	600	800	40	1.1s / 40,145.3	4.7s / 818,868.4	4.3	20.4
4	1000	1500	80	2.0s / 68,807.6	12.7s / 2,551,821.7	6.4	37.1
5	2000	4000	80	7.0s / 196,292.1	49.2s / 7,639,212.5	7.0	38.9
6	2000	4000	300	10.5s / 186,074.8	147.4s / 23,404,727.8	14.0	125.8
7	2000	6000	400	16.4s / 311,941.4	296.7s / 48,110,168.0	18.1	154.2

Table 1. Characterization of the random benchmarks by groups

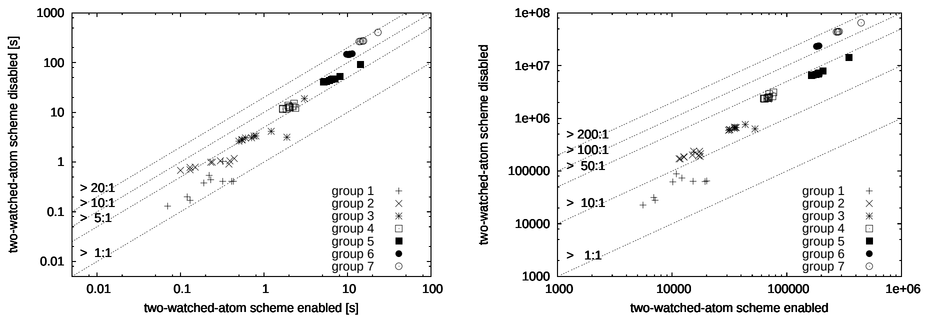

Fig. 4. Performance impact of the two-watched-atom scheme: runtime in seconds (left) and number of clause evaluations (right)

case of solving mixed Boolean and non-linear arithmetic formulae. The results point out that the essential factor for the performance of the 2WAS is the clause size (cf. groups 5&6) rather than, e.g., the number of clauses (cf. groups 2&3 and 4&5). The reason obviously is that, compared to the naive approach, the more atoms a clause contains, the more clause visits can be saved. Formulae of group 7 consist of a non-trivial size of $6000 \cdot 400 = 2,400,000$ atoms. Here, the 2WAS shows performance gains of more than one order of magnitude wrt. the runtime and a few orders of magnitude concerning the number of clause evaluations.

4 Conclusion and future work

In this paper we discussed the generalized 2WLS for mixed Boolean and arithmetic formulae as implemented in our constraint solver HySAT. We explained the consequences of the fact that in the iSAT algorithm the watched atomic formulae have a more complex structure than in the propositional case. We presented our solutions to deal with those peculiarities, among them a two-level-watch scheme which allows an efficient update of the watch lists when a watch-pointer is removed from a complex atom. Experimental results show that the performance gains achieved by 2WAS in the setting of arithmetic constraint solving are similar to those observed in purely propositional SAT.

One of our main focuses for future work is to investigate decision heuristics. This issue was extensively studied in the context of purely Boolean SAT solvers, where the choice of an adequate decision heuristics turned out to be instrumental to scalability and stable performance across a range of problems. In general, however, there is no decision heuristics which fits to all kinds of problems.

We are confident that most of the decision heuristics from SAT can be adapted to our framework, and will turn out to be beneficial. Moreover, in our more general setting of mixed Boolean and arithmetic problems, contrary to purely propositional SAT solving, the domain of an arithmetic real-valued or integer variable is no longer two-valued s.t. even for a single variable, there is a variety of possible splits to select from. Thus, in addition to the common aspects underlying decision heuristics in propositional SAT, like the activity of a variable during solving, we will develop new decision heuristics and take account of the type of a variable (Boolean, real-valued, integer), its current interval width, etc., when optimizing the HySAT tool through dedicated splitting heuristics.

Preliminary experiments involving a generalized variant of the *variable state independent decaying sum* (VSIDS) decision heuristics [MMZ$^+$01] show that there is a huge potential in this field. As in the propositional case, we introduce activity counters (for the interval borders of the variables) and prefer interval borders and variables with high activity when splitting intervals. On some benchmarks, the speed-ups for the generalized VSIDS are considerable (e.g. by a factor of 40), while on the other hand, there are also examples on which VSIDS performs worse than naive splitting (e.g. 10 times more runtime). Thus, in future work we are going to identify which decision heuristics are suitable for which subclasses of formulae.

Another very interesting point is that we are not confined to traverse the search space by splitting intervals only, which means dividing the space in parallel to the axis. We can also split the search space by deciding some arithmetic expressions, e.g. deciding $x > y + z$ or $y \leq 5 \cdot x$, which are not necessarily atoms of the formula but newly generated ones.[9]

Acknowledgements

The authors are very grateful to the anonymous reviewers for their helpful comments.

References

[BG06] F. Benhamou and L. Granvilliers. Continuous and interval constraints. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint*

[9] Note that we then allow also inequalities like, e.g., $x > y + z$ besides equations like, e.g., $x = y + z$. However, interval deduction via ICP and evaluation of the truth value can be smoothly adapted.

Programming, Foundations of Artificial Intelligence, chapter 16, pages 571–603. Elsevier, Amsterdam, 2006.

[BPT07] A. Bauer, M. Pister, and M. Tautschnig. Tool-support for the analysis of hybrid systems and models. In *Proceedings of the 2007 Conference on Design, Automation and Test in Europe (DATE'07)*, Los Alamitos, CA, April 2007. IEEE Computer Society.

[CK03] D. Chai and A. Kuehlmann. A fast pseudo-boolean constraint solver. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 830–835, New York, NY, USA, 2003. ACM Press.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[DP60] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.

[ES03] N. Eén and N. Sörensson. An Extensible SAT-Solver. In *6th International Conference on Theory and Applications of Satisfiability Testing*, 2003.

[FH03] M. Fränzle and C. Herde. Efficient SAT engines for concise logics: Accelerating proof search for zero-one linear constraint systems. In A. Voronkov M. Y. Vardi, editor, *Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2003)*, volume 2850 of *LNCS, subseries LNAI*, pages 302–316. Springer, 2003.

[FHR$^+$07] M. Fränzle, C. Herde, S. Ratschan, T. Schubert, and T. Teige. Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure. *JSAT Special Issue on SAT/CP Integration*, 1:209–236, 2007.

[GJM06] I. P. Gent, C. Jefferson, and I. Miguel. Watched literals for constraint propagation in minion. In F. Benhamou, editor, *CP*, volume 4204 of *LNCS*, pages 182–197. Springer, 2006.

[KB03] G. Katsirelos and F. Bacchus. Unrestricted nogood recording in csp search. In F. Rossi, editor, *CP*, volume 2833 of *Lecture Notes in Computer Science*, pages 873–877. Springer, 2003.

[LSB07] M. Lewis, T. Schubert, and B. Becker. Multithreaded SAT Solving. In *12th Asia and South Pacific Design Automation Conference*, 2007.

[LSTV07] C. Lecoutre, L. Sas, S. Tabary, and V. Vidal. Recording and Minimizing Nogoods from Restarts. *JSAT Special Issue on SAT/CP Integration*, 1:147–167, 2007.

[MMZ$^+$01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. of the 38th Design Automation Conference (DAC'01)*, June 2001.

[RCOJ06] G. Richaud, H. Cambazard, B. O'Sullivan, and N. Jussien. Automata for Nogood Recording in Constraint Satisfaction Problems. In *Proceedings of the CP 2006 First International Workshop on the Integration of SAT and CP Techniques*, pages 113–127, 2006.